Archimedes

C-51 Cross-Compiler Kit

For

8051 Microcontroller Development.

Eighth Edition

Archimedes Software, Inc., San Francisco, CA.

Disclaimer

The information in this document is subject to change without notice. While the information contained herein is assumed to be accurate, Archimedes Software, Inc. assumes no responsibility for any errors or omissions.

Copyright Notice

Trademarks

Archimedes and Microcontroller C are trademarks of Archimedes Software, Inc.
MS-DOS is a trademark of Microsoft Corp.
UNIX is a trademark of AT&T and Bell Labs.
VAX, MicroVAX, VMS and Ultrix are trademarks of Digital Equipment Corporation.

Edition

Eighth Edition, October 1987.

Preface.

Welcome to Archimedes Microcontroller C!

This manual describes how to use the Archimedes C-51 Cross Compiler Software Development Kit for MCS-51 8-bit microcontrollers.

Product Overview. The Archimedes C-51 kit is for software development of any 8051 proliferation chip. The kit includes a C-compiler, assembler, linker, librarian as well as a number of libraries. The object code produced by the compiler can be downloaded to a PROM-programmer, emulator, simulator, symbolic debugger or the target system.

The C-compiler is based on the traditional Kernighan and Ritchie C-standard. More importantly, it is also an implementation of the proposed ANSI-standard for C-compilers. A set of C libraries include functions as I/O and string support as well as advanced IEEE 32-bit floating point support.

The assembler allows flexibility in speeding up time-critical sections of code as well as accessing certain hardware features. The linker links modules of code written in C and/or assembler and relocates code into absolute memory addresses.

The librarian is used to create and manage libraries. The linker's many output formats support most emulators and PROM-programmers.

Manual Overview. This manual is not a primer. It assumes a basic knowledge of C and assembly language programming. The manual is written to complement other literature, such as Intel's Microcontroller Handbook, which you already may have.

Preface

Table of Contents

TUTORIAL

T.1 What this Tutorial Covers

This section of the Archimedes C-51 manual is designed to familiarize the first-time user with the basic principles of operation for getting up and running in a minimum amount of time. We will present an overview of C-51 and walk through an example program to illustrate the use of some of the more important features. We will explain how to compile C-51 code, link it with assembly language functions and create files for downloading to an emulator or PROM programmer. Other topics include modifying the C "startup" and putchar (character output) modules, writing interrupt handlers, debugging techniques, emulator support and common problems and their solutions.

The compiler (C-51), linker (XLINK), assembler (A8051), and librarian (XLIB) that make up the C-51 kit each have their own Chapters that follow this Tutorial. As these contain much more detailed descriptions of C-51 use, the Tutorial does not attempt to cover every subject in full. We suggest that you review these other sections after you have read the material presented here.

If you have used an Archimedes C compiler for another microcontroller (68HC11, 6301, Z80, etc.), this section will serve as an introduction to details that are specific to the 8051 family. Please note that you will find a great deal of similarity among the different products.

Because a thorough discussion of the 8051 MCU (microcontroller unit) and the C language is beyond the scope of this manual, we assume that the reader is familiar with the following subjects:

- The architecture and features of the 8051 family of microcontrollers. It is also necessary that you have at least a basic knowledge of 8051 assembly language.

- The "C" language itself, as defined by Kernighan and Ritchie, or by the proposed ANSI X3J11 "C" standard (see the references listed below).

Start out by reviewing the tutorial and the (colored) release pages (at the back of the manual). If you are familiar with C and assembly programming as well as the 8051 architecture you may start programming after you have also read through the C-chapter. If you are a true beginner, you should review the complete manual carefully before you get started. The manual is organized as follows:

Chapters:

Tutorial
1. C
2. Assembler
3. Linker
4. Librarian

Appendices:

A. Programming Hints
B. Libraries
C. Proliferation Chip Support
D. C Error Messages
E. Assembler Error Messages
F. Linker Error Messages
G. Librarian Error Messages
H. Other Development Tools Support
I. Code Example
J. Memory Maps

Please review the 'Tutorial' for other recommended literature. For your convenience, the software is not copy-protected. We trust your integrity in not making any illegal copies and only using the software on one machine at a time.

Again, welcome to Archimedes Microcontroller C!

We recommend the following books as references and learning aids for these subjects:

The Intel Microcontroller Handbook, or equivalent data book for your 8051-based microcontroller (80515, 80C451 etc.): This describes the chip-level features and use of the 8051 family of microcontrollers.

The Intel MCS-51 Macro Assembler User's Guide: Serves as a complete reference to the 8051 instruction set.

Intel Literature Department
3065 Bowers Avenue
Santa Clara, CA 95051
Ph. (800) 538-1876

The C Programming Language, by Kernighan and Ritchie: The de-facto standard (and for years, the only) reference book on the "C" language. Also known as the "white book".

Prentice-Hall, Inc.
Englewood Cliffs
New Jersey, 07362

The "C" language series from Que Corporation are all highly recommended. Beginners will especially benefit from the first two titles:

The C Programming Guide, 2nd Edition
The C Self-Study Guide
Advanced C: Techniques and Applications
Common C Functions
C Programmers Library
Debugging C

Que Corporation
P.O. Box 50507
Indianapolis, IN 66250
Telephone orders: 1-800-428-5331

Please note that the official X3J11 ANSI standard document for the "C" language has not been finalized as of this writing. However, a summary of the constructs and keywords that are specific to ANSI C may be found in Section 1.23 of the C-Compiler Chapter.

T.2 Standard Files and File Types

The actual files included with your kit will vary with the version number -- refer to the release notes (colored pages in the manual) for a current directory listing.

The general file types that are used and created with C-51 are listed below. Please note that not all occurances of these file types can be found on the distribution media (e.g., no .LST files are included):

```
*.EXE   EXECUTABLE FILES (E.G., C-51.EXE, THE COMPILER)
*.HLP   HELP FILES (FOR XLINK AND XLIB)
*.C     C LANGUAGE SOURCE FILES
*.H     C HEADER FILES, NORMALLY #INCLUDED IN C SOURCES
*.INC   ASSEMBLY LANGUAGE INCLUDE FILE (SOURCE)
*.R03   LIBRARIES AND OTHER RELOCATABLE OBJECT FILES
*.S03   8051 ASSEMBLY LANGUAGE SOURCE FILES
*.XCL   EXTENDED COMMAND FILES FOR COMPILER AND LINKER
*.LST   DEFAULT LIST FILE TYPE USED BY COMPILER OR LINKER
*.A03   DEFAULT OUTPUT (ABSOLUTE) FILE TYPE USED BY LINKER
*.BAT   MS-DOS BATCH FILES FOR INSTALLATION OR COMPILATION
*.DOC   EXTRA DOCUMENTATION TEXT FILES
```

T.3 Installation

If you have not already done so, please follow the installation instructions for your version of the software kit as described in the Release Notes (colored pages) of the manual.

T.4 Other Development Tools

You may use any ASCII text editor, such as Brief, VEDIT, EMACS or WordStar, to create and modify C-51 source and control files. Please note that when using a general-purpose word processing program, such as WordStar, to edit source code, be sure to set the editor in its "non-document" or plain ASCII mode of operation.

The XLINK linker directly supports a large number of output formats so that C-51 absolute object code can be downloaded to almost any standalone development board, hardware emulator or PROM programmer for testing and creation of EPROMs. 8051 object code can also be tested on your host system using an 8051 software simulator. Full symbolic debugging capability is available with most emulators and simulators. See Appendix H (or the colored release notes) for list of development tools that are supported with C-51. If the tool you are using is not shown, please call Archimedes Technical Support -- it may be compatible with one of those listed or support for it may have been recently added.

You may also want to make use of a native ANSI C compiler and debugger for your (PC) host when developing code for C-51. The PC-tools include Microsoft C with CodeView, Borland Turbo C, Aztec C, and others. One of the advantages of ANSI-compatible C code is that it may be easily ported from one target to another with minimal changes as long as standard rules of portability are observed (use standard I/O functions as much as possible, modularize programs, etc.).

This allows the "generic" (hardware independent) portions of a C-51 program to be compiled and tested entirely on the host system using a native compiler and debugger (e.g., Microsoft C and CodeView). Basic program logic, math functions, user interface routines, data structures, etc., can be verified this way. When the generic code is operating properly on the host system, it can be combined with 8051-specific portions and then compiled and linked with C-51. Testing of the final program on the target system with an emulator or other debugging tool is thus minimized (this is often the most time-consuming part of development).

Appendix A, Programming Hints, contains some suggestions as to how C-51 code can be written so as to be testable on a host system.

T.5 Archimedes C and the 8051

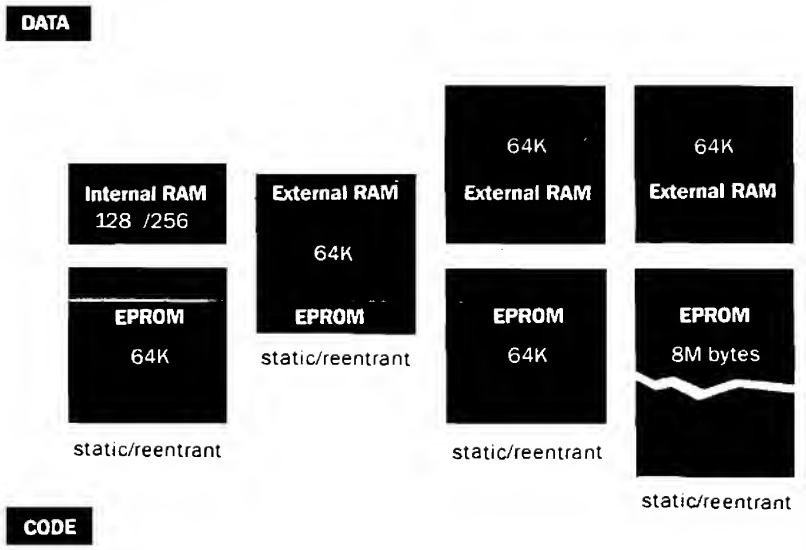
The C-51 language is a complete implementation of the proposed ANSI C standard, which is actually a refinement of the K&R (Kernighan and Ritchie) definition. A set of the most common library functions is included to facilitate portability with other C compilers. In addition to these "standard" C features, Archimedes has added a number of extensions to support the development 8051 microcontroller-based applications. The list below summarizes these additions; more details on each one can be found in the example code to follow and in the C-Compiler Chapter.

Multiple Memory Models: The 8051 is unique in that it can access three separate physical memory spaces: Internal RAM (data), external RAM (data) and internal/external ROM (code). C-51 supports four basic "memory models" to best match the hardware configuration of your target system:

small:	internal RAM only
medium:	64K combined external RAM and PROM
large:	64K of PROM and 64K external RAM
banked:	banked-switched PROM (1MB+), 64K ext. RAM

The memory models are further divided into "static" and "reentrant". In the reentrant models, memory for all local "auto" variables is dynamically allocated from the stack when a function is called and freed when the function exits. In the static models, auto variables are forced by the compiler to reside in a static memory area. The reentrant models are used when an interrupt handler is written in C or when a function is called recursively (i.e., it calls itself). The static models are used when execution time is the critical factor as access to the stack is minimized.

The diagram below illustrates from left the different different memory models supported by C-51, i.e. small, medium, large and banked:



While it is possible to use the small memory models with the 8051/31 chips (128 bytes of internal RAM), we recommend that you use the 8052/32 (or other compatible) chip with 256 bytes of RAM (especially if you are writing interrupt handlers in C or have many levels of function nesting in your program).

PROMable Code: The compiler can generate PROMable code in which the initializers for static variables will be stored in ROM and then copied over into their RAM variables at run time. The -P compiler option is used to enable PROMable code.

On-Chip Hardware Access: C-51 includes a powerful set of "in-line" functions for direct access to 8051 SFRs (Special Function Registers) and internal RAM while using any of the memory models. Both bit and byte-oriented functions are included.

The byte-oriented functions read and write a single byte of internal RAM or an SFR. "ADDRESS" must be a constant in the range of 0-255:

input unsigned char input(ADDRESS);
output void output(ADDRESS, unsigned char data);

The bit-oriented functions are used to test, set and clear a single bit located in an SFR or in bit-addressable RAM (20-2F hex). "BIT_ADDRESS" must be a constant in the range of 0-255 (not all addresses are valid):

read_bit
unsigned char read_bit(BIT_ADDRESS);
set_bit
void set_bit(BIT_ADDRESS);
clear_bit
void clear_bit(BIT_ADDRESS);
read_bit_and_clear
unsigned char read_bit_and_clear(BIT_ADDRESS);

The symbols that represent the 8051 SFRs and their control bits are pre-defined in a file called IO51.H, so that they may be referred to by name:

```
clear_bit(EA_bit);        /* disables all interrupts by clearing EA */
```

Please note that these functions act like C-51 keywords, or extensions to the C language. They are not called or declared as normal functions but generate efficient in-line code for directly manipulating hardware with a minimal amount of software overhead. For example, the following line of C code produces just 3 8051 instructions and executes in under 5 microseconds (12Mhz clock). Your own symbolic names (ALARM, DOOR, etc.) could be substituted for the pre-defined bit names:

```
/* if P1.0 and P1.1 are ones, set P1.7 to one */

if(read_bit(P1_0_bit) && read_bit(P1_1_bit)) set_bit(P1_7_bit);
```

For compatibility, the special compiler option (-e) must be used to enable the recognition of these inline functions.

Direct Access of External Data and Code Memory: C-51 includes 3 more in-line functions that provide direct access to external data (RAM) and internal or external code memory (ROM). These functions read or write a byte at the variable "address":

read_XDATA unsigned char read_XDATA(unsigned int address);
write_XDATA void write_XDATA(unsigned int address);
read_CODE unsigned char read_CODE(unsigned int address);

The read_XDATA and write_XDATA functions are especially useful where the small memory model is selected (for its inherent code efficiency) but external RAM is used for a buffer, or there are external memory-mapped I/O devices. Normally, small model C programs cannot access external memory directly. These functions allow easy, efficient access without the need for assembly language calls.

In the large memory model, the programmer does not normally have direct access to data that is stored only in code memory (ROM). This is because all C variables and strings are assumed to be in DATA memory (RAM). The read_CODE function is especially useful where direct access to ROM-only data is needed. Appendix A contains a programming example using read_CODE.

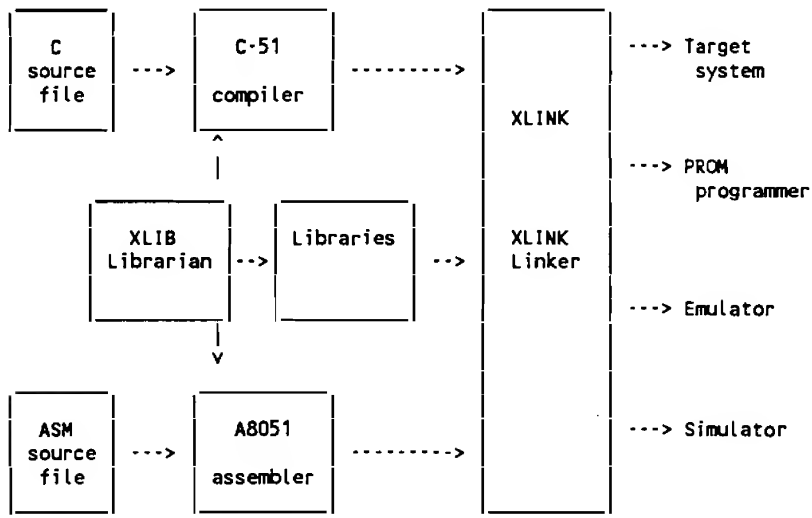
Like the internal bit and byte manipulation functions, the -e compiler option must be used to enable these functions as well.

T.6 8051 Proliferation Chip Support

A number of microcontroller vendors offer specialized devices based on the Intel 8051 "core", including the 8344 (Intel), 80C451 (Signetics), 80535, and 80515 (Siemens). The difference between these chips lies in their various combinations of memory, I/O ports and on-chip peripherals (timers, A/D, etc.) and not in their instruction sets. Therefore, support for a new device can be easily added by modifying the IO51.H file, where the 8051's I/O ports and registers are defined. C-51 includes a definition file for one proliferation chip, the Siemens 80515 (see the file IO515.H).

T.7 The Development System

The C-51 kit consists of a compiler, assembler, linker and librarian programs. The diagram below illustrates how these are used together to develop an 8051 application. (An explanation of each program follows).



Compiler: The C-51 compiler is a single program that reads C source (filetype .C) and header (.H) modules as input and directly produces a relocatable object file (.R03) as output, which can then be processed by the linker. Because of this single-pass design, a separate assembly phase is not required. The output file is in an Archimedes-proprietary format called UBROF (Universal Binary Relocatable Object Format) that contains code, data and symbolic information.

By using compiler switches, various listing files can be produced as well, including one with intermixed C and assembler statements (-L -q options, .LST filetype). A straight assembler source file (.ASM) can also be generated (-a) that allows the programmer to modify and re-assemble the compiler output to further optimize the code, if required.

In addition to the standard typechecking required by ANSI C, the C-51 compiler includes a very useful built-in program verifier (-

g option) similar to the Unix "LINT" utility. It checks your source code for potentially erroneous (though legal) conditions such as unreferenced local variables, unreachable code or constant array indexing that is out of range. When enabled, this option also causes typechecking information to be included in the output file for all external references, so that the linker can verify that the interface between modules is consistent. For example, say source file MOD1.C contains a function declaration:

```
int some_function(int a, long b)
{       .
        .       /* body of function */
```

And MOD2.C contains a prototype and call that do not match the real function as declared in MOD1.C (the second parameter should be a "long" value):

```
extern int some_function(int a, int b);  /* prototype */
main()
{       int i,j;
        some_function(i,j);              /* call to function */
}
```

When these two modules are linked together, XLINK would print an error message (assuming both modules were compiled with the -g switch):

```
Warning[6]: Type conflict for external/entry
some_function, in module mod2 against external/entry in
module mod1
```

Assembler: The A8051 macro assembler reads a standard Intel 8051 assembler source file as input (.S03 default filetype) and creates a relocatable object file in the UBROF format as output (.R03). A list file with cross-references may be also generated (.LST). It operates as a single, two-pass program.

The assembler is provided so that time-critical sections of code may written in 8051 assembly language to minimize the amount of software overhead. For example, a timer interrupt handler that is invoked every few hundred microseconds should be coded in assembler. If desired, the C-51 compiler output (with the -a switch) can be reassembled after further optimization, though this is purely optional.

A8051 is compatible with the Intel 8051 assembler, ASM-51, with the exception of a few directives (pseudo-ops), the use of ACALLs and AJMPs, and the macro facility. These differences are explained in the Assembler Chapter.

Linker: XLINK is a flexible linker/loader that reads as input relocatable UBROF-format object files (.R03 filetype) created by the the compiler or assembler. The -Z linker command is used to set the base addresses for the various code and data segments to match the ROM and RAM memory maps of your target system.

As output, XLINK produces an absolute object file with optional symbols or symbol file in one of a number of different formats to support most emulators, simulators and PROM programmers, including: Intel HEX, Intel AOMF (for the ICE-51 or 5100 emulator), Ashling, Microtek, and many others. A listing file (.LST) can also be produced that contains a module map, a segment map, and a cross-reference listing of global symbols. XLINK is normally operated in a "batch" mode where it reads its options and input file names from a command file (.XCL filetype).

Two types of modules can be loaded with XLINK: A "program" module is one that will always be loaded (i.e., placed in the output file image) when the name of the file in which it is located is processed by XLINK. The CSTARTUP module (which contains essential run-time initialization code) and most user-written C modules will have the program attribute.

The other type is a "library" module, which is loaded only when it contains an entry (a function or global variable name) that has been referenced by another module. (This approach minimizes run-time library overhead). This has the effect of only loading modules from a library file that are really needed, thus minimizing memory usage and and reducing link time. The standard C-51 library files CL8051.R03, CL8051S.R03 and CL8051B.R03 (for large/medium, small, and banked models, respectively) all contain modules that have the library attribute. CSTARTUP is the only module in the library file that has the program attribute, so it is normally the only one that is always loaded when the library file is processed.

XLINK does not allow linking C-51 or A8051 object modules with those produced by compilers or assemblers from other vendors (e.g., Intel's PL/M-51 or ASM-51). However, most ASM-51 (or other 8051 assembler) programs can be easily converted and reassembled with Archimedes A8051. (See the assembler chapter for more details).

Librarian: The XLIB utility is used to manage the standard C-51 library files and to create user libraries. XLIB operates on relocatable object files (filetype .R03) and permits listing, renaming, replacing and deleting modules, segments and entries. A module may also be given a "program" or "library" attribute with XLIB.

T.8 Putting Together the Example Program

We will now demonstrate the use of the C-51 kit by walking through the example program provided in source form on the distribution diskettes. This program illustrates the concepts and procedures that will help you create your own C-51 programs.

In creating a C-51 program for the first time, these steps should be generally followed:

1. Examine and modify the CSTARTUP module. Replace the original version with the modified one in the library file (this step may be optional).

2. If the standard character output functions are needed (printf and/or putchar), you must modify the PUTCHAR.S03 assembler routine to support your hardware configuration. Replace the putchar supplied in the library (which is for the Intel SDK-51 development board) with the modified version. Do the same for character input (getchar) if this function is needed.

3. Edit and compile your C source file(s). If desired, you can compile and test the "generic" (non-8051 specific) portions of the program with a host-resident compiler (e.g., Microsoft C with CodeView).

4. Edit and assemble any assembly language source file(s).

5. Modify one of the provided linker command files (.XCL filetype) to match your ROM/RAM memory map, select the program modules to be loaded and set the output file format to that used by your emulator, simulator or PROM programmer.

6. Download the absolute object file to your hardware emulator, simulator, target board or PROM programmer for testing.

We will now build the example program using these steps as a guideline. The complete source files are included on the distribution media and listed in Appendix I; only partial listings are shown below.

T.8.1. Modifying and Assembling CSTARTUP

A number of critical C-51 system functions are located in the CSTARTUP assembler module, so we will examine it in detail.

- Allocates the internal and external stack segments
- Initializes the startup 8051 register bank
- Initializes the internal and external (virtual) stack pointers
- Makes calls to initialize the data segments
- Makes a call to the main() program function
- Adds in user-specific 8051 interrupt vectors
- Adds in user-specific hardware initialization code

The file CSTARTEX.S03 is a version of CSTARTUP that has been modified for our example application. A timer interrupt vector and handler were added and the internal stack size was change from 20 to 45 bytes. These are typically the type of changes that you would need to make.

```
        NAME            CSTARTUP; module name

$DEFEM.INC      ; Include file to define memory model used (large)

        PUBLIC  exit
        EXTERN  _R              ; Register bank no. (0,8,16 or 24)
        EXTERN  main            ; main is C program entry point
        EXTERN  ring_bell       ; C interrupt handler
```

The NAME statement does not only name the module but gives CSTARTUP the program attribute. This means it will always be loaded when the file in which it is located (the library) is processed by the linker. The MODULE statement is used instead of NAME when a LIBRARY (conditional load) is desired.

The "$DEFEM.INC" statement tells the assembler to include the definition file for the selected memory model, which in our case is going to be large. This file controls the conditional assembly of various parts of the module. One of the following files must be included:

```
DEFSC.INC       for the small, or single chip, model
DEFEM.INC       for the medium or large "expanded" models
DEFMB.INC       for the banked model
```

Since we have selected the large model, we will only examine those portions of CSTARTUP that apply to that model (see Appendix I or the disk file CSTARTEX.S03 for a complete listing).

The following code allocates memory for the internal and external stacks. In the expanded models, the internal stack (data segment ISTACK) is used for storage of return addresses (from LCALLs), temporary workspace for some C runtime routines (mainly math functions) and for saving register contents in an interrupt handler. The external stack (data segment XSTACK) is used for dynamic storage of local auto variables and for passing formal parameters to functions. Here we allow 45 bytes for ISTACK (the recommended size if interrupt routines are written in C) and 512 bytes for XSTACK.

```
        IF              single_chip = 0 ; Expanded models only
        RSEG            XSTACK          ; Segment for external stack
        DS              512             ; Good size to start with,
xstack_end:                             ;  change if needed
        ENDIF

        RSEG            ISTACK          ; Segment for internal stack
stack_begin:
        IF              single_chip
        DS              150
        ELSE                            ; For expanded models...
        DS              45              ; 45 bytes of internal stack
        ENDIF                           ;  for C interrupt handlers
```

The CSTART segment is normally the first code segment, and is linked at location zero, so the SJMP to init_C is the first instruction executed after a reset. Location 0Bh is the Timer 0 vector location, so we must place an SJMP here to our interrupt handler. If you are using other interrupt sources (external, serial port, etc.), the vectors would be added here in a similar fashion. Refer to the *Intel Microcontroller Handbook* for a list of the 8051 interrupt sources and their vectors.

```
        RSEG            CSTART  ; Code segment used by CSTARTUP only
startup:                        ; Must be linked to be at location zero
        SJMP            init_C
        ORG             startup + 0Bh   ; ORG locates vector at location
        SJMP            handler         ;  for Timer 0
                                        ;  add other vectors here...
```

The init_C initialization code is ORGed to begin following the end of the interrupt vector area:

```
        ORG             startup + 30H   ; Code starts after 8051 vectors
init_C:
        MOV             A,#_R           ; _R is the register bank,
        MOV             C,ACC.3         ;  defined at link-time
        MOV             PSW.3,C
        MOV             C,ACC.4
        MOV             PSW.4,C         ; working bank is now in PSW
        MOV             SP,#stack_begin ; 8051 stack grows low to high

        EXTERN  ?SEG_INIT_L17   ; CL8051 Library routine to
        LCALL   ?SEG_INIT_L17   ;  initialize data segments

        MOV             R6,#HIGH(xstack_end)    ; Init. external stack
        MOV             R7,#LOW(xstack_end)     ; ** DO NOT REMOVE **

                        ; Add user hardware initialization code here...
```

C-51 normally uses register bank 0 for its working registers. This can be changed at link-time by redefining the symbol "_R". The internal stack in the 8051 grows from low to high addresses, so the SP register is initialized to point to the beginning of the ISTACK segment. The address of ISTACK is determined at link-time, but is normally set to start at location 7 (so the first PUSHed item is stored at location 8).

?SEG_INIT_L17 is a run-time library function that initializes the data segments. Static initializers, located in ROM, are copied into their corresponding variables or strings, located in RAM. Any uninitialized variables are also set to zero, according to the ANSI standard.

The LCALL to this function MUST NOT be removed when the large or banked model is used. This is because large model code assumes that all data elements reside in the data (RAM) memory space, and ?SEG_INT_L17 is needed to initialize strings and other C constants located in RAM. However, this call may be removed in the small and medium models if statically initialized variables are not needed.

Note: Functions that begin with "?" are C-51 "helper" routines contained in the libraries. Calls to these functions are transparently generated by the compiler to augment the rather limited 8051 instruction set. They are never called directly by the user from a C-51 program, but some are used in assembly language routines (see PULSE.S03).

Registers R6 and R7 are used as a 16-bit pointer to manage the external (virtual) stack. **The above code which initializes R6:R7 to point to the end of the XSTACK segment MUST NOT be removed.**

It is sometimes necessary to perform hardware initialization in assembler or enable interrupts before entering the main C program. This is normally the place to insert such code. Please note, however, that if your hardware design includes bank-switched RAM, you must enable the proper bank BEFORE the call to ?SEG_INIT_L17. (C-51 does not directly support bank-switched data memory (RAM), but this can be done explicitly by the user, of course.)

Finally, an LCALL is made to the C program entry point, the main() function. DPTR is set to zero to indicate that no parameters are being passed to it. When the main function terminates, the "SJMP $" puts the 8051 in an infinite loop (most microcontroller applications never exit but run continuously). A jump to a monitor or debugger program could be placed there instead.

```
        MOV             DPTR,#0         ; DPTR = 0 means no parameters
        LCALL   main                    ; User C code entry point main()
exit:                                   ; The exit() function goes here
        SJMP    $                               ; Loop forever...
                                        ; Or add jump to a
                                        ;  monitor/debugger
```

The initial handler for the Timer 0 interrupt is shown below. As the ring_bell() interrupt service routine is written in C, all working registers must be pushed onto the internal stack before it is called.

```
handler:
        PUSH            ACC                     ;  Save  all  working
registers...
        PUSH            B
        PUSH            PSW
        PUSH            DPH
        PUSH            DPL
        PUSH            0
        PUSH            1
        PUSH            2
        PUSH            3
        PUSH            4
        PUSH            5
        MOV             DPTR,#0         ; 0 parameters passed to C
        LCALL   ring_bell               ; C interrupt service routine
        POP             5               ; Restore the registers...
        POP             4
        POP             3
        POP             2
        POP             1
        POP             0
        POP             DPL
        POP             DPH
        POP             PSW
        POP             B
        POP             ACC
        RETI

        ; Handlers for other interrupt sources can be added here....
```

It is sometimes necessary to write an interrupt handler entirely in assembler for the sake of efficiency. In these cases, only those registers that are used by the handler need to be saved.

The CSTARTEX.S03 module is assembled and put in the CL8051.R03 library (replacing the original version) as follows:

```
C>A8051 CSTARTEX

C>XLIB
*DEF-CPU 8051
*REP-MOD CSTARTEX CL8051
*EXIT
```

CSTARTUP does not have to be loaded from the library -- it can be linked in explicitly as a standalone module if desired. However, be sure to delete the original CSTARTUP module from the library first to prevent it from being automatically loaded as well (see the Librarian Chapter).

T.8.2. The PUTCHAR and GETCHAR Routines

Like CSTARTUP, the putchar (character output) and getchar (character input) functions must usually be custom tailored for your target hardware if you intend to use standard character I/O. Since the printf (formatted print) function calls putchar, it will work when putchar works. If your application does not call for standard character I/O, there is no need to modify these functions and you may skip this section.

The versions of putchar and getchar that come installed in the standard libraries are written to support the Intel SDK-51 development board, which uses an external (off-chip) UART for its console I/O. The assembler source code for these functions can be found in the PUTCHAR.S03 and GETCHAR.S03 files.

The assembler source for a minimal version of putchar which uses the 8051's on-chip serial port is shown below. Additional code must be written to initialize the port's baud rate and data format. This can be done in assembler in CSTARTUP or in C using the input and output functions. For detailed information on programming the on-chip serial port, please refer to the Intel Microcontroller Handbook.

```
; Minimal version of putchar for on-chip port  (not on disk)
;   called with the character to transmit in register R3
;   C prototype:  int putchar (int val)      (see STDIO.H)
;
        MODULE  putchar
        PUBLIC  putchar
        RSEG            CODE
putchar:
        JNB             TI,putchar              ; wait for the TI bit
        CLR             TI                      ; reset TI bit
        MOV             SBUF,R3                 ; write character
        RET
        END
```

After making your changes, the PUTCHAR.S03 module should be assembled and put in the library replacing the distribution versions:

```
C>A8051 PUTCHAR           (source_file)

C>XLIB
*DEF-CPU 8051
*REP-MOD PUTCHAR CL8051
*EXIT
```

If you explicitly link in your modified putchar module (instead of putting in the library), be sure to delete the old version from the library first (in XLIB, the DELETE-MODULES command).

T.8.3. The EXAMPLE.C Program

We will now look at our EXAMPLE.C program which is contained on the distribution diskette and listed in its entirety in Appendix I. Most C-51 programs will follow this same general structure, which has the form:

```
#include header files
#define constants and C-macros
declare global and external variables
prototypes for external functions

function1
        declare local variables

function2
        declare local variables
.
.
main program body
        declare local variables
```

The first section of the example program is shown below:

```
#include <stdio.h>      /* required header for printf and putchar */
#include <io51.h>       /* required header for output, bit_set, etc. */
#define TRUE 1
#define FALSE 0
#define BELL 7          /* ASCII terminal bell */
#define SIZE 8190        /* size of array for sieve routine */

char flags[SIZE+1];     /* array for sieve function */
int count = 1;           /* pass counter */

extern int pulse(int count, char value);  /* assembler function */
```

The IO51.H file contains #defines for 8051 bit and byte locations that are used to symbolically access the Special Function Registers (SFRs). This header file should be included whenever the input, output, or bit-manipulation functions are used.

The last line is an ANSI prototype for an external assembly language function contained in the file PULSE.S03, which will be assembled and linked with our main program. This prototype defines the exact number and type of formal parameters and the return value so that the compiler and linker can check the interface between modules. All functions located outside of the calling source module (i.e., in another file) should be declared in this manner.

Likewise, a global variable that is to be referenced in more than one source file should be declared as "extern" in all modules except the one where it is defined and its storage space allocated. For example, if function(s) contained in another source file need to have access to the global variable "count", the other file should contain the following external declaration:

```
        extern int count;          /* now accessible to this entire source
file */

        int function1(int i)
        {
                if(i = count)
                        return(1);
                .
                .
        }
```

If there are a large number of global variables, their declarations can be placed in a header (.H) file which is then #included in the source modules that need it.

The first sub-function in our program is set_timer, which sets up and starts Timer 0. The output and set_bit inline functions are used to directly access the timer and interrupt registers from C. Refer to the *Intel Microcontroller Handbook* for register usage and programming:

```
void set_timer()   /* initialize Timer 0 to generate interrupts */
{
  output(TMOD,1);         /* Timer-0: mode 1 (16-bit timer) */
  output(TH0,0x4C);       /* load 4C00h for interrupt every 50 ms */
  output(TL0,0);
  set_bit(TR0_bit);               /* set Timer 0 run control bit */
  set_bit(ET0_bit);               /* set interrupt mask bit ET0 */
  set_bit(EA_bit);                /* enable interrupts bit EA */
}
```

Please note that the inline functions output and bit_set DO NOT need to be declared in a header file or elsewhere as they are treated like extensions to the C language (when the -e compiler option is used).

The next section of our example program is the Timer 0 interrupt handler that is called from CSTARTUP:

```
void ring_bell()   /* Timer-0 interrupt handler, rings the CRT
                    * bell approximately every 3 seconds; called
                    * from vector in CSTARTUP module */
{
  static int intr_ctr = 0;  /* counter for number of interrupts */
  output(TH0,0x4C);      /* reload timer again, hi byte */
  output(TL0,0);         /* low byte */
  if(++intr_ctr == 60)   /* only do every 60 interrupts */
  {
    putchar(BELL);       /* ring the bell, or do some useful task */
    intr_ctr = 0;
  }
}
```

The next function, do_io, further demonstrates the use of direct hardware access using C-51:

```
#define BIT_VAR 0        /* 1st bit address in internal RAM (20.0 hex) */
void do_io()     /* demonstrate 8051 I/O functions */
{
  char c;
    /* write and read Port 1 */
  output(P1,0x0F);
  printf("\nPort 1: %02X\n",input(P1));
        /* Write and read an absolute external data address,
     * such as a UART or other memory-mapped I/O device.
     * "(char *)" casts the int constant to a char pointer. */
  *(char *)0xE000 = c;   /* write to device at E000h */
  c = *(char *)0xE000;   /* read */
  c = read_XDATA(0xE00); /* another way, for small model */
  printf("Data at E000h = %02Xh\n",c);
    /* just for the sake of it, print a byte of CODE memory */
  printf("Location 0 in code memory is: %02X",read_CODE(0));
        /* show use of bit-addressable RAM variables */
  if(read_bit(P1_0_bit) || read_bit(P1_1_bit))
    set_bit(BIT_VAR);
}
```

In the expanded models, the use of C pointers allows easy access to absolute external memory locations, as would be needed for a data-memory-mapped I/O device such as a UART. However, ANSI C does not permit the direct use of integer constants as pointers, or the assignment of integers to pointers. To do this, a cast of the integer constant (e.g., the UART's address) must be performed first, as shown above.

In the small model, external data memory is accessed using the read_DATA and write_DATA functions. These may also be used in the expanded models instead of the pointer method (they are equally efficient).

C-51 does not directly support the 8051's bit-addressable RAM. However, bit-variables can be allocated manually and tested, set and cleared using the inline bit functions, as shown above. The first location that can be used is the LSB of location 20h, which is bit-address 0. The last is the MSB of location 2Fh (bit address 7Fh). If using these locations, be sure to locate the internal stack (segment ISTACK) above bit-addressable RAM, as shown in the link process description to follow.

The main program sets up Timer 0 and simply calls the other functions continuously (the sieve benchmark routine is not described here):

```
void main()   /* Main EXAMPLE.C program */
{
  set_timer();          /* turn on timer and interrupts */
  printf("Timer enabled, bell should ring every 3 seconds.\n\n");
  while(1)                      /* do forever */
  {
    printf("EXAMPLE.C sample program, pass %d.\n",i++);
    do_io();               /* display RAM, Port1, etc. */
    printf("\nPulsing Port 1.7 (8051 pin 8)\n");
    pulse(200,0xA5);      /* call assembler routine pulse P1 */
    sieve();              /* do the Sieve benchmark */
  }
}
```

T.8.4. Compiling the Program

The example program is compiled with C-51 as shown below. We suggest that you try this on your computer at this time (compile and unmodified copy of EXAMPLE.C for now):

```
C>c-51 example -m0 -V -e -g -L -q       (source_file options...)

        Archimedes 8051 C-Compiler V2.01A/M02
      (c) Copyright Archimedes Software Inc. 1987

Errors: none
Warnings: none
Code size: 660      (actual size of example may vary)
```

The first parameter in the command line following the C-51 program name is the name of the C source file; a filetype of .C is assumed, so it is normally left off. The EXAMPLE.C file should be in the current DOS directory; a directory path may be used to specify an alternate directory for the source file (e.g., \c-51\src\example). The standard header files (STDIO.H, IO51.H, etc.) should be in the current directory or in the directory defined by the C_INCLUDE environment variable (see the Installation instructions in the Appendix). Alternately, the search directory for #include files can be specified on the command line with the -I option. Please note that the search directory name must have a trailing backslash (\), as in:

```
c-51 example -m0 -V -e -g -P -L -q -I\c51\include\
```

The -m0 option specifies the large reentrant memory model (see section 1.5 Memory Models in the C-chapter for more details). Please note that different memory models use different c-libraries (cl8051, cl8051s, cl8051b) that need to be selected in a in a linker command file (see section T.8.6 and section 1.13 in the C-chapter for more details).

Two output files are created by this run of C-51: EXAMPLE.R03, which contains the relocatable object code

(Archimedes proprietary UBROF format), ready to be linked; and EXAMPLE.LST, a C/assembler list file. As C-51 is entirely memory-resident, no temporary files are created and compile speed is very high.

The "Code size" refers to the amount of inline code produced by the compiler. When the program is linked with the standard library, a number of runtime "helper" routines will be added, as will other C and assembler modules and user-called library functions (putchar, printf, etc.). The final code size is determined by examining the linker's segment or module map.

The compiler "switches" that follow the source file name are the most-commonly used compiler options, so we will discuss each one. A full list of compiler options appears in the C-Compiler Chapter, or simply type "C-51" with no parameters to display the compiler help message. These options can appear before or after the source file name in the command line.

-m0 Selects the large reentrant memory model (the default, so this option could have been left out). The large model supports up to 64k of code (EPROM) and 64k of data (RAM) memory. This is the most commonly-used model where external memory is used. The reentrant version was selected here because our interrupt handler was written in C. The other models are:

-m1 large static
-m2 medium reentrant
-m3 medium static
-m4 small reentrant
-m5 small static
-m6 banked reentrant
-m7 banked static

Refer to the C-Compiler Chapter for a more complete discussion of selecting the proper memory model for your application.

-V Enable the "verbose" compiler error messages, which pinpoints (if possible) the error position in the offending source line:

```
printf("Location 0 in code memory is: %02X",read_CODE(0);
.....................................................^
"example.c",66  Error[48]: ')' expected
```

-e Enables the recognition of C-51 extensions to the ANSI C language. This option allows the following inline C functions to be used: input, output, read_bit, set_bit, clear_bit, read_bit_and_clear, read_XDATA, write_XDATA, read_CODE.

-g Enables the strict global typechecking feature which warns about the use of certain legal, though possibly dangerous, constructs in your code. We strongly suggest the use of this option as it helps eliminate many types of common C errors. It also enables the generation of typing information in the output file so the linker can check the interface between modules.

-P Generates PROMable code, where static initializers for variables are copied from PROM to the variable in RAM at run time by the ?SEG_INIT_L17 function (see CSTARTUP). In the large and banked models, this option can be left out, as statically initialized variables are always handled this way because all data elements must be located in data memory (RAM).

-L
-q These two options, normally used together, cause the generation of a list file named EXAMPLE.LST that contains a mixture of C code (-L) and the corresponding assembly assembly-language statements (-q). This file is useful for debugging purposes as well as just studying the compiler's output. A small portion of EXAMPLE,LST is shown below. We suggest that you study the complete list file after compilation to get an idea how C-51 produces code, uses segments, and allocates data:

```
.
.
65          /* read a byte of CODE memory */
66        printf("Location 0 in code memory is:
%02X",read_CODE(0));
\   00B8  7B00            MOV     R3,#0
\   00BA  7A00            MOV     R2,#0
\   00BC  8A83            MOV     DPH,R2
\   00BE  8B82            MOV     DPL,R3
\   00C0  E4              CLR     A
\   00C1  93              MOVC    A,@A+DPTR
\   00C2  FB              MOV     R3,A
\   00C3  7A00            MOV     R2,#0
\   00C5  120000          LCALL   ?PUSH_R2_R3_L17
\   00C8  7B26            MOV     R3,#LOW(?0007)
\   00CA  7A00            MOV     R2,#HIGH(?0007)
\   00CC  900004          MOV     DPTR,#4
\   00CF  120000          LCALL   printf
67                  /* show use of bit-addressable RAM variables
*/
68          if(read_bit(P1_0_bit) || read_bit(P1_1_bit))
\   00D2  209003          JB      P1.0,?0008
\   00D5  309102          JNB     P1.1,?0009
\   00D8            ?0010:
\   00D8            ?0011:
\   00D8            ?0008:
69              set_bit(BIT_VAR);
\   00D8  D200            SETB    32.0
\   00DA            ?0009:
70        }
\   00DA            ?0012:
\   00DA  020000          LJMP    ?LEAVE_L17
.
.
```

You will notice in the listing some labels that are LCALLed that begin with a "?". These are C-51 "helper" routines that are needed to implement various runtime functions that cannot be produced efficiently with inline code. The compiler automatically produces these calls transparently to the programmer. Some of these functions can be useful in writing assembly language programs that will interface with C.

Another option, -a, produces a pure assembly language source file that can be reassembled if necessary, though there is usually little need for this.

T.8.5. An Assembler Function

While most microcontroller applications can be coded completely in C, it is often necessary to write small portions in assembler for the sake of code efficiency (execution speed). This is most often true with timer and other high-speed interrupt handlers. The PULSE.S03 assembler source file demonstrates how to create a C-callable assembler function.

```
        ; int pulse(int count, char value);
        ;
        NAME            pulse
        PUBLIC  pulse
        EXTERN  ?POP_R3_L17             ; library "helper" function
        RSEG    CODE                    ; use the CODE segment
pulse:
        CLR             P1.7            ; turn off/on P1.7
        NOP
        SETB            P1.7
        DJNZ            R3,pulse        ; using only lo-byte of count
        LCALL   ?POP_R3_L17             ; get second parameter into R5
        MOV             P0,R3           ;  write to P1
        MOV             R3,P3           ; return with int value of
        MOV     R2,#0                   ;  Port 3 in R2-R3 pair
        RET

        END
```

This routine takes the first formal C parameter, int count, and pulses the 8031's pin 8 by that many times (it only uses the low-order byte). The second parameter, char value, is then written to Port 1.

For the sake of efficiency, the first parameter passed to a function (C or assembler) is contained in the 8051 registers R2-R5, as follows (except for "union" and "struct" -- see the C-Chapter):

```
R3     8-bit char values
R2-R3  16-bit int and pointer values (R3 being the low-order byte)
R2-R5  32-bit long and float values
```

The second, and any additional, parameters are passed on the external C stack and must be popped off using one of the ?POP routines, as shown above. Refer to the C-Compiler Chapter 1.12 for a full list of these ?POP and ?PUSH routines used for C/assembler interface. Please note that this function is written for a large or medium model C program. In the small model, there is no external stack, so the second and additional parameters are passed on the internal stack; see the code example for the "uadd" function in section 1.10 of the C-Compiler Chapter.

Incidentally, using single-parameter C and assembler functions as much as possible is one way to help optimize large and medium model code produced with C-51. This minimizes the amount of external stack manipulation, which involves a fair amount of runtime overhead.

The value returned to the calling C program should be placed in the registers R2-R5, according to the table above. In our example function, we return the int value of Port 1 with R2 (the high byte) set to zero.

There is no need to save any of the 8051 registers when writing assembler functions, **with the exception of R6-R7, which must not be used**. The PULSE.S03 file is assembled as follows:

```
C>A8051 PULSE PULSE      (source_file list_file)

    Archimedes 8051 Assembler V1.80/MD2
  (c) Copyright Archimedes Software 1985

  Errors:  None          #########
  Bytes:   17            # pulse #
  CRC:     405D          #########
```

Two files are created by the assembler: PULSE.R03, the relocatable object file, ready to be linked with the main C program; and PULSE.LST, a list file.

T.8.6. Linking the Example Program

XLINK is now used to link together the EXAMPLE.R03, PULSE.R03 and CL8051.R03 (large/medium library) relocatable object modules into a final, absolute program. Try linking the example program under control of the EXAMPLE.XCL command file, as shown:

```
C>xlink -f example

        Archimedes Universal Linker V3.04A/MD2
    (c) Copyright Archimedes Software Inc. 1987

Errors: none
Warnings: none
```

This link operation will produce an absolute object file, EXAMPLE.A03, and a cross-reference/map file, EXAMPLE.MAP.

When linking C code, XLINK should always be used with the aid of a command file (filetype .XCL) that contains the input file names and the switches. XLINK can accept all of its input file names and options on the command line, but it should only be used in this manner when linking small assembly language programs. The EXAMPLE.XCL command file that comes with C-51 is shown below (less some of the comment lines):

```
-! EXAMPLE.XCL:  XLINK command file for example program -!
-c8051
-Z(IDATA)ISTACK=7
-D_R=0
-Z(CODE)CSTART,RCODE,CODE,CDATA,ZVECT,CONST,CSTR,CCSTR=0
-Z(XDATA)DATA,TEMP,IDATA,UDATA,ECSTR,WCSTR,XSTACK=0
example
pulse
cl8051
-x
-l example.map
-z
-FAOMF8051
-o example.a03
```

Please note that the kit comes with several different command files (*.xcl) which use different c-library functions depending on which memory model you are using. The right c-libraries must be used with the right mmemory models.

The most-often-used XLINK options are explained below. For a complete list and detailed description, refer to the Linker Chapter of the manual, or type "XLINK" with no parameters for a help message. The order of the switches in the file is of no importance as the linker scans them all first before processing the input files. These commands may also be entered on the XLINK command line from DOS:

-! This is used to bracket comments, which may extend several lines. Be sure to leave a space after the last comment and the trailing "-!".

-c Used to define the CPU type, which is 8051 in our case (XLINK supports many different CPU types).

-Z This command is used to define the type and base address of the various code and data segments used by C-51. There are three lines using the -Z command in EXAMPLE.XCL:

```
-Z(IDATA)ISTACK=7
```

This tells the compiler that the segment ISTACK is a segment with a type of "IDATA" (internal 8051 data) and that it should start at location 7. Since the 8051 stack grows upward, the first item placed on the internal stack would be at location 8. The size of the ISTACK segment is determined in CSTARTUP.S03.

You are free to locate the internal stack anywhere in internal memory, with the exception of addresses 0-7, where register bank 0 resides (or which ever register bank has been selected).

```
-Z(CODE)CSTART,RCODE,CODE,CDATA,ZVECT,CONST,CSTR,CCSTR=0
```

The next -Z command determines where the various code segments used by C-51 will reside in memory (ROM). The (CODE) designator at the beginning of the line, which is optional, tells the linker that these segments have a type of "CODE" so it can verify that the segment names that follow all have this same type. If you attempted to link a segment that had a type of "DATA", an error message would be generated.

The segment names listed here (CSTART, RCODE, CODE, etc.) are the pre-defined names that C-51 uses for various types of executable code and constant (read only) data. If you examine the list-file output from the compiler using the -L and -q options, you can see how these are used. A list of these segments and their functions can be found in the C-Compiler Chapter.

It is best to ALWAYS USE this standard list of segment names when linking C code, even if you believe a segment will not be needed by your program.

The number at the end of the list is a hexadecimal address that determines where the first of these segments will be linked. In our case, we want CSTART to be located at zero so that the CSTARTUP code will be executed following a processor reset. The second segment (RCODE) will start immediately following the end of the first, and so on, up to the last code segment.

The base address for these code segments will normally be the address of where the EPROM or ROM starts in your target system.

```
-Z(XDATA)DATA,TEMP,IDATA,UDATA,ECSTR,WCSTR,XSTACK=0
```

This command determines where the data segments used by C-51 will reside in memory (RAM). The (DATA) designator at the beginning of the line, which is optional, tells the linker that these segments have a type of

"DATA". The segment names listed here (DATA, TEMP, IDATA, etc.) are the pre-defined names that C-51 uses for various types of read/write storage, including variables, strings, and external stack. If you examine the list-file output from the compiler using the -L and -q options, you can see how these are used. A list of these segments and their functions can be found in the C-Compiler Chapter.

Again, it is best to ALWAYS USE this standard list of segment names when linking C code, even if you believe a segment will not be needed by your program.

Like the code segment list, the base address for the data segments appears at the end. This is usually the start of RAM memory in your target system. In our case, RAM also starts at location zero. This is valid, since we are using the large model, which supports separate data and code address spaces of 64k bytes each.

-D This command is used to define a symbol at link-time. We set the special symbol "_R" equal to zero here to select register bank 0 for use by C-51. The value 8, 10 or 18 (hex) may also be used to select bank 1, 2 or 3 instead. This option is useful when some other piece of software running alongside -51 code (such as a real-time executive) needs to use register bank 0. However, all C-51 code in a given program must use the same bank.

example
pulse
cl8051

These are the names of the example files to load and link; a filetype of .R03 is assumed. A module with a "program" attribute will always be loaded from a file when it is processed by XLINK. Since C modules are "programs" by default, the module EXAMPLE will be force loaded. Since the PULSE module contains a NAME statement, it, too, will be considered a "program" and will be loaded.

However, the standard C-51 library file, CL8051, contains many modules, most of which have the "library" attribute, so only those modules that have entries (functions) that are referenced somewhere will be loaded. This includes user-invoked C functions (printf, putchar, etc.) and runtime helper routines (? functions). The one exception in CL8051 is the module STARTUP, which will always be loaded since it contains a NAME statement, which gives it the "program" attribute. Therefore, only those library functions needed for a given program will be included in the link process.

-l
-x These options tell the linker to generate a complete cross-reference listing file for the program including an absolute memory map of where all the segments and modules are located. This is a useful tool for determining RAM and ROM usage and for debugging the final program. The -l option is used to specify the output filename (EXAMPLE.MAP); if omitted, the listing will be sent to the screen.

A section of the EXAMPLE.MAP file created by the linker is shown below. A section of the module map is followed by the segment map:

```
FILE NAME : example.r03
PROGRAM MODULE, NAME : example

SEGMENTS IN THE MODULE
======================
CODE
  Relative segment, address : 0745 - 09D8
          ENTRIES                ADDRESS     REF BY MODULE
          sieve                   0822       Not referred to
          set_timer               0745       Not referred to
          ring_bell               0762       CSTARTUP
          main                    098F       CSTARTUP
          do_io                   07A0       Not referred to
```

```
.
.
 SEGMENT      START ADDRESS     END ADDRESS   TYPE  ORG  P/N  ALIGN
 =======      =============     ===========   ====  ===  ===  =====
 ISTACK          0007    -       0033        rel   stc  pos   0
 CSTART          0000    -       007E        rel   stc  pos   0
 RCODE           007F    -       0744        rel   flt  pos   0
 CODE            0745    -       14C2        rel   flt  pos   0
 CDATA           14C3    -       14C4        rel   flt  pos   0
 ZVECT           14C5    -       14C5        rel   flt  pos   0
 CONST           14C5    -       14C5        rel   flt  pos   0
 CSTR            14C5    -       14C5        rel   flt  pos   0
 CCSTR           14C5    -       1605        rel   flt  pos   0
 DATA            0000    -       0000        rel   stc  pos   0
 TEMP            0000    -       0000        rel   flt  pos   0
 IDATA           0000    -       0001        rel   flt  pos   0
 UDATA           0002    -       2000        rel   flt  pos   0
 ECSTR           2001    -       2141        rel   flt  pos   0
 WCSTR           2142    -       2142        rel   flt  pos   0
 XSTACK          2142    -       2341        rel   flt  pos   0
```

-z This switch tells the linker NOT to perform a check for overlapping segment addresses, a condition that occurs normally in the 8051 architecture because of its separate code, external data and internal data memory spaces.

-F
-o The -F option is used to determine the output file type. The "AOMF8051" format chosen here creates a binary file with symbolic information that can be downloaded to an Intel (or compatible) in-circuit emulator. Many other symbolic and non-symbolic formats are supported, including INTEL-STANDARD hex code, so that almost any in-circuit emulator, software simulator or PROM programmer can be used with C-51. See the Appendix H for a complete list, or call Archimedes if your development tool is not shown.

The output file name is determined with the -o option. Here, the absolute output file is named EXAMPLE.A03.

T.8.7. Downloading and Testing

The final output file, EXAMPLE.A03, can now be downloaded and tested on your emulator or target board, if desired. Refer to the documentation provided with your emulator or PROM programmer for specific instructions.

Please note that if the output file type specified in the -F option is one that contains symbolic information, only global-type symbols will be included (function names, global static variables, PUBLICed assembler labels etc.). If you need to see local static (compiler-produced) symbols, the C-51 "-j" option should be used when compiling a C program.

A hint: When debugging C code on a PC-hosted emulator or development board, you can use a "pop-up" notepad program (like SideKick, from Borland International) to load the C-51 listing or linker map file so it can be easily referenced while you are working. Remember that the compiler listing does not contain absolute addresses for functions or static data -- you must use it in conjunction with the link map or your emulator's symbol table.

T.9 Glossary of Terms

The terms included here are those that are specific to, or especially important in, C-51. Please refer to a C textbook for general C language definitions.

ANSI-C — The "official" definition for the C language proposed by the American National Standards Institute, number X3J11. Not yet approved as of this writing. See Section 1.22 in the C-Compiler Chapter for a summary of ANSI-C features.

auto — A C variable declared locally within the body of a function is by default an "auto" variable (its storage class). The memory space for an auto variable is dynamically allocated from the stack

when a function is called, and de-allocated when the function exits. Auto variables are only supported in the reentrant memory models; they are converted to "static" by the compiler if a static model is used. Also see: static.

bit-address

The 8051 architecture supports direct addressing of certain bit-locations in internal RAM and in some of the Special Function Registers. The first 128 bit addresses (0-7Fh) are used to access the internal bit-addressable RAM (at byte addresses 20-2Fh). Bit addresses 80-FFh are used to access bits within some of the SFRs. See the file IO51.H and the Intel Microcontroller Handbook for details.

code

In respect to the architecture of the 8051, "code" refers to the physical memory space that contains executable instructions for the CPU or read-only data. Code memory is usually ROM or EPROM. Fetches or reads of code memory are strobed by the chip's "/PSEN" signal. 8051-family chips can access up to 64k of code in their expanded modes of operation. Also see: data.

cross-software

Software development tools that run on one type of computer or processor and produce code for another, different, type of processor. C-51 runs an 8088/86-based (PC/AT) or VAX-based host processor and produces code for the Intel 8051 microcontroller; it includes a cross-compiler and a cross-assembler.

data

In respect to the architecture of the 8051, "data" refers to the physical memory space that contains read/write data. Code memory is usually RAM. Reads amd writes to data memory are strobed by the chip's "/RD" and "/WR" signals. 8051-family chips can access up to 64k of data in their expanded modes of operation. Also see: data.

global — A variable that is declared outside of a function body is a global (vs. local) variable (its scope). Global variables are also static, that is, they reside in a fixed location in memory. Only global variables (and functions) are normally included in the symbol table output by the linker.

helper functions — These are C-51 library routines that implement various low-level runtime functions that cannot be efficiently implemented with compiler-generated inline code (e.g., long division). Calls to these routines are transparently and automatically generated by the compiler. All helper function names are prefaced with a "?" (e.g., ?L_DVMD_L15).

host — Refers to the computer/processor that is used as the development system to run the C-51 compiler, assembler, linker, etc. C-51 supports PC/AT and VAX host computers. Also see: target.

in-line functions — Those C-51 functions that generate inline code rather than involve a subroutine call. The C-51 functions input, output, set_bit, etc., are used to directly access on-chip hardware (ports, timers, etc.) from C. Since these are treated as extensions to the C language, the -e compiler option must be used to enable their use.

K&R — Refers to the "old" definition of C as contained in the book, *The C Programming Language*, by Brian Kernighan and Dennis Ritchie of Bell Labs. The differences between K&R and ANSI C are listed in the C-Compiler Chapter, section 1.22. Also see: ANSI.

keyword — Any word that has a special meaning to the C compiler and, therefore, cannot be used as a variable, function or constant name.

library — A C or assembler module may have a "library" or "program" attribute. A library module is one that will only be loaded by the linker if it is needed (i.e., contains an entry that is referenced by another module that has already been loaded). A C program can be given the library attribute with the -b compiler switch, while the MODULE statement makes an assembler program a library module. Most of the modules contained in the standard library files (CL8051?.R03) have a library attribute. Also see: program.

macro — A macro is a single C or assembler statement that the macro pre-processor (a part of the compiler or assembler) expands into a series of in-line statements that perform an often-used task. C macros are created with the #define statement and MACRO is used for assembler macros.

module — A function or group of functions that makes up part or all of a C-51 program. With the C-51 compiler, a single module is produced by each C-source file that is compiled, and they share the same name. In the assembler, a single source file may contain multiple modules. A module is made up of one or more functions and data elements.

program — A C or assembler module may have a "program" or a "library" attribute. A program module is one that will always be loaded by the linker when the file it belongs to is processed. By default, a C module has the program attribute. An assembler module is given a program attribute with the NAME statement. The only module in the standard library files (CL8051?.R03) that has the program attribute is CSTARTUP. Also see: library.

reentrant — A function that calls itself, or a C-51 function that is interruptable by another C-51 function. The "reentrant" memory models (-m0, -m2, -m4 and -m6) are used if reentrancy is required. Reentrant functions use the stack for local variables.

segment — A segment is a group of code or data elements that are all related in some functional way, so they may be linked into a program together as a whole. For example, the segment RCODE will contain C-51 instructions that must reside in a resident (non-banked) read/only memory. C-51 uses a number of pre-defined segments (see the C-Compiler Chapter for a list). User-defined segments can also be created (with the RSEG assembler directive).

SFR — An 8051 Special Function Register, which is the means by which a program access the hardware features of the chip (timers, ports, interrupt masks, etc.). Special C-51 in-line functions are used to access the SFRs from C. Also see: in-line functions.

startup routine — (CSTARTUP) This is the code that the 8051 processor executes immediately following a hard reset or power-on. It (CSTARTUP.S03) consists of runtime initialization code essential to the operation of C-51 programs.

static — A C variable located in a fixed location in memory that is active throughout the program (it is not de-allocated, as are auto variables). This also refers to the "static" memory models (-m1, -m3, -m5, -m7), where all variables are forced to be static for code efficiency. See also: auto, reentrant.

target — Refers to the processor and its associated hardware that the compiler and assembler generate code for (i.e., the Intel 8051 or compatible). Also see: host.

UBROF — The Universal Binary Relocatable Object Format, which is an Archimedes-proprietary format used for output files from the C-compiler and assembler. XLINK reads UBROF files as input.

1.1 C - Introduction

The 'C' language, invented by Brian W. Kernighan and Dennis M. Ritchie (K&R), has become a major system-programming language for applications ranging from simple controller-type programs to complete operating systems. Its unique portability assures that code can be moved between radically different architectures with considerably less effort than is possible with other 'standard' languages such as Pascal and FORTRAN.

After its birth in the early seventies the C language has evolved creating certain 'dialects' which have sometimes been incompatible with the original C definition (K&R). In order to regain control of the C language, the American National Standards Institute (ANSI) has set up a group of industry software experts to update the language definition. The Archimedes C development team is involved as an observer in this group and the compiler described in this document closely follows the proposed draft standard. The *official* standard, however, will not be finalized until 1988 so there *may* be some changes in the definition which compiler manufacturers will have to conform to.

Most of the ANSI additions are minor changes to enhance portability with one notable exception: function prototyping. Function prototyping allows C users the option of full parameter checking, as in Pascal, coupled with the C language assignment conversion rules.

The Archimedes C is based on the K&R-standard with ANSI-enhancements implemented.

1.2 C-Chapter Overview

This manual is not a primer and it has been written to complement other documentation. The objective of the C-chapter is to explain the specifics of using the Archimedes C Cross-compiler for microcontroller development relative to a standard native C-compiler, rather than exploring the basics of the C-language.

In other words, this manual builds on any earlier C-experience you might have and you are recommended to have your favorite C book handy as a general C language reference. As a tutorial *The C Programming Language*, by the original C authors Kernighan & Ritchie, is suitable though its language definition differs slightly compared to the proposed ANSI-standard. Also, see **Tutorial** for other recommended C-literature.

(The proposed ANSI-standard documentation is available from the ANSI committee. However, it is not recommended reading since it is written with the sole purpose of clearly defining a standard). Section 1.22 provides a quick reference to the most important language enhancements through the ANSI-standard.

To fully utilize the capabilities of the Archimedes C-compiler you also need to familiarize yourself with the other chapters in the manual. Appendix D summarizes all the C error and warning messages. You will also find several of the other appendices useful.

1.3 C-51 Overview

The Archimedes C-compiler is a portable program almost completely written in C. It is based on a memory-intensive design (no temporary files or overlays) that results in a very speedy and also relatively simple system. The compiler executes as a single program that directly generates relocatable code in Archimedes's proprietary UBROF format (Universal Binary Relocatable Object Format).

Several code generating options are available to best support the requirements of different microcontroller designs.

The "expanded mode" options are particularly suited to 8031-type designs using external memory. All data is assumed to reside in external RAM with internal RAM used for the C run-time system and user-written assembly routines. (The assembler can of course access user defined data and memory locations within the internal RAM). The bank-switched memory models allow you to design larger 8051 systems.

In the "minimum single-chip static mode" all data resides in internal RAM. The static allocation of data is an alternative to a traditional stack. This mode speeds up execution but limits reentrant (interruptable and recursive) code.

Bit manipulation and symbolic access in C to on-chip features are supported via special 8051 in-line functions (which are extensions to the standard C language). The compiler performs code optimization by default. The use of built-in speed (-s) or code size (-z) compiler-switch options allow further improvements.

In addition to the type-checks required by the ANSI-standard, Archimedes has also integrated a substantial part of the functions provided by the UNIX C program verifier 'LINT' directly in the compiler. With this facility (activated by the command line switch -g) programs using the older K&R-form of function declarations/definitions can be checked during compilation of a module (module = file). Unlike LINT which only works at the source-level, the Archimedes system leaves the interface-checking to the linker. This has the advantage of making checks on library function usage feasable, even when only available in object form. Also, identifiers have no less than 255-character identifiers through-out the system (compiler, assembler and linker).

The compiler output is to be processed by the Archimedes universal linker XLINK to obtain an executable file. Linker output is usually fed to the target system, PROM-programmer or emulator.

A C-compiler option of generating assembler code may be used to facilitate debug. It has options of generating full cross-reference lists as well as paginated list-files. The compiler has advanced error recovery and diagnostic systems. Arrows indicate the exact source code error location and an error message describes the error detected.

The Archimedes C-compiler is invoked in a manner similar to a UNIX compiler but with the addition of a simple built-in help facility.

C source code can be written with any standard ASCII full-screen editor.

1.4 Data Representation

The C compiler supports all ANSI C basic elements. Variables are always stored with the most significant part located at low memory address. "Float", "double" and "long double" are all implemented in the IEEE 32-bit single precision format. Plain "char" is equivalent to "unsigned char" in this implementation (which can be changed with the **-c** switch). The table below shows the size in bytes of the different objects.

char	short	int	long	float	pointer/address
1	2	2	4	4	2

Please note that the ANSI-standard specifies short integer as a 2-byte data-type. If you prefer to have an 1-byte short integer (in your whole program) you can accomplish this by using the C-language #define pre-processing directive and the schar data type (e.g. #define shrt schar).

As in any standard C, integer or character hex constants may be entered in the "0xA800" format.

Register variables as a storage class is recognized but currently ignored by the compiler due to the register requirements of the C run-time.

"Signed char" or "schar" is an ANSI-standard 1-byte element. It can be used for both characters and numbers range (-127 to +127).

Floating point supports the four basic arithemtic operations (+,-,*,/). The following advanced math functions are also available: atan, asin, acos, tan, cos, sin, sqrt, exp, pow, exp10, log and log10. To ease the use of the mathematical functions a header file **math.h** is also featured in the package. (Also, see App. B).

Float numbers are stored as follows with the most significant bit to the left (i.e. the "Sign" bit is the most significant bit):

[Sign = 1 bit] [Exponent = 8 bits] [Mantissa = 23 bits]

In other words, the four registers R2 through R5 contain the following information:

```
SEEEEEEE EMMMMMMM MMMMMMMM MMMMMMMM
```

The "Sign" bit is 1 if it's a negative number. The "Mantissa" is normalized.

A normalized non-zero number X has the form of:

```
X = [(-1)** S] * [2**(E-127)] * [1.F] where
S = Sign bit
E = 8-bit exponent biased by 127.
F = X's 23-bit fraction which together with an implicit leading 1
yields the significant digit field "1.---".
```

1.5 Overview Memory Models

The C-51 compiler supports several hardware configurations, or memory models, to best meet the requirements of different microcontroller designs. (Similar to 8086 small and large model).

Expanded memory models - large and medium. There are two expanded memory models, i.e. large and medium. In both the large and medium models, the C variables and the run-time stack reside in external data memory. The large model supports microcontroller applications with 64K of code *and* 64K of data ans has options of reentrant (-m0) and static (-m1) models. The medium model supports microcontroller applications with a combined total of 64K code and data. It requires that the Program Status ENable signal (/PSEN) must be AND-ed together with the Read Data signal (/RD), to create a uniform 64k address space. The medium model supports both reentrant (-m2) and static (-m3) models.

Small memory model. This model supports 8051 configurations using only internal RAM. C variables and the run-time stack reside within internal RAM. In the small memory model it is recommended to use a chip with at least 256 bytes of internal RAM (e.g. 8032, 8052). Most practical C-programs require this due to the stack-intensive C-language. Supports reentrant (-m4) and static mode (-m5).

Banked memory model. The bank-switched memory model is identical to the large model when it comes to variable allocation. However, it extends the accessible *code* area beyond the 64K limit with the help of a simple memory mapping scheme. It support reentrant (-m6) and static options (-m7).

All memory models offers two approaches on how to allocate variables - reentrant and static. In the reentrant modes (**-m0**, **-m2**, **-m4** or **-m6**), all local "auto variables" are allocated and deallocated dynamically, i.e. they reside on a stack required to support recursive or reentrant functions. In the static modes (**-m1**, **-m3**, **-m5** or **-m7**), all function-level variables are forced into static memory with the exception of function arguments which are always on the stack.

Code from all modes may be partially tested on the host computer using resident C compilers and debuggers. This approach assumes you follow common C rules for writing machine-independent code. Of course, code with functions relying on 8051 I/O specific parts may not be verified in this way.

Memory Model	Banked Reentrant	Banked Static	Expanded Reentrant	Expanded Static	Small Reentrant	Small Static
Typical chip	8031/32	8031/32	8031/32	8031/32	8051/52	8051/52
External RAM	Yes	Yes	Yes	Yes	No	No
Code Area	>1M	>1M	64K	64K	64K	64K
Recursion	Yes	No	Yes	No	Yes	No
C Interrupt Routines	Yes	Limited	Yes	Limited	Yes	Limited
C Variable RAM Area	Ext. RAM (64K)	Ext. RAM (64K)	Ext. RAM (64K)	Ext. RAM (64K)	Int. RAM (256)	Int. RAM (256)
Rel Speed	Low	Low	Low	Medium	Medium	High
Rel Code Compactness	Medium	Medium	Medium	Medium	Medium	High
C-Libraries	cl8051b.r03		cl8051.r03		cl8051s.r03	

Please note that you need to select the proper memory model, c library function file and the corresponding linker command file (*.xcl) depending on which memory model that you are using. The small, banked modes, and expanded modes have different C run time systems, thus *modules generated by the different modes cannot be combined.* If modules compiled with incompatible memory models are linked together, the linker will flag either duplicate definitions or unresolved externals. Modules from the different expanded modes, both reentrant and static, may though be linked together (although this is not recommended). Review Appendix J for an overview of the memory organization in different memory models.

To choose the model that is the best for you please review the table and the following sections describing the different memory models. You need to review the requirements for:

- Internal vs. external RAM.
- Use of lots of local auto, recursion, function call nesting, printf, sprintf, malloc and free library functions.
- 64K CODE & DATA vs. 64K CODE & 64K DATA.
- Tying two pins together (/PSEN and /RD).
- Time-critical code.
- Recursive code (lots of interrupts).
- Minimize external RAM (due to PROM in large model).

If a statically allocated object is prefixed with "const" this object will be put into the ROM section only (CONST segment). This is valuable when a lot of static data structures must be created as it conserves RAM space. This is an ANSI supported feature.

```
typedef struct
  {
      declarators.....
  } s;

const s table[]=
  {
     initializers.....
  };
```

Please note that the "const" attribute will only have this effect in the "medium" and "small" memory models. Review section 1.16 on how to use the in-line function read_CODE to access ROM only data in the large memory model.

Since the "small" and expanded memory models are incompatible, the user-configurable as well as user-written assembly language routines must know what model to use. In order to ease things a bit the routines supplied in this package reference an include file **defmm.inc**. This file is supposed to be created by copying **defsc.inc** (small), **defem.inc** (expanded) or **defemb.inc** (banked) to **defmm.inc**. Conditional assembly does the rest. Study **cstartup.s03**, **putchar.s03** and **getchar.c** for more info.

1.6 Small Memory Model

The small single-chip models are primarilly suited for "classic controller applicaitons" (i.e. code that mainly involves simple decision logic and I/O). This is due to that internal RAM is rather limited in the [current] 8051 family. This means that extensive use of local variables, recursion or function call nesting should be avoided. Due to RAM limitations the C library functions **isxxxx**, **malloc** and **free** are of little use in the small memory model. The formatting routines **printf**, **sprintf** however, can be used if *numeric* fields are less than 25 bytes long (i.e. a "%030d" specification will fail while "%30d" will execute well).

The small model library resides in the file **cl8051s.r03**. **The small single-chip and the expanded (medium and large) models have different C run time systems; thus modules generated by single-chip and expanded modes cannot be combined. If incompatible memory models are linked, the linker will flag either duplicate definitions or unresolved externals.** (You are recommended to use a chip with 256 bytes of on-chip RAM (e.g. 8032/52) for advanced C-programs (due to C stack requirements).

With **write_XDATA** and **read_XDATA** (see section 1.14), external data memory can be accessed even in the small memory model. In the small memory model, if a statically allocated object is prefixed with "const" this object will be put into the ROM section only (CONST). (App. J for small memory model organization).

1.7 Large/Medium Expanded Memory Models

There are a total of four different expanded memory models available - large reentrant (default, -m0), large static (-m1), medium reentrant (-m2) and medium static (-m3). All these models use a 'virtual stack' located in external RAM.

The large memory models make it possible to generate code for a system with 64K of code *and* 64K of data. The medium models require that *the Program Status ENable signal (/PSEN) must be AND-ed together with the Read Data signal (/RD) in order to create a uniform 64K system* (to avoid a conflict between the C run-time system needs and the [8051's] separate instructions for accessing external RAM and external or internal ROM/EPROM.

The large and medium models are similar. Both use the same library routines. However, code generated for the different models should not be combined.

Code density (i.e. size of generated code) is the same for both memory models, but variables will in the "large" memory model also occupy some space from the code section according to the following:

- Uninitialized variables will only be allocated in [external] DATA memory.

- **Initialized variables will take the same space from both the CODE and [external] DATA memory. The initializer constants are stored in CODE memory and copied into the DATA memory at start-up (completely transparent to the user).** This also applies to C string literals. (This is due to that C variables are always accessed via MOV X instructions rather than through a mix of MOV X and MOV C instructions. The latter would require inefficient 17-bit addresses in various routines).

- The 'const' directive has no effect on variable allocation in the large memory models.

Examples (applies to all non-auto variables):

```
int i = 10;           /* 2 bytes CODE and 2 bytes DATA */
char *p = "ABC";      /* 2+4 bytes in both CODE and DATA
                      2 bytes for *p, 4 for "ABC"  */

int j;                /* 2 bytes in DATA memory only */
char big_buf[10000]; /* 10000 bytes in DATA area */
```

Note that static and extern variables without explicit initializers are automatically set to zero at start-up according to the C standard.

In the medium memory model, if a statically allocated object is prefixed with "const" this object will be put into the ROM section only (CONST). **Code is always PROMable in the "large" memory model** (i.e. the -P command line option is redundant). Code generated by the "large" memory model will also be usable in a 64K (PSEN+RD ANDed) system, if linked in such a way that code segments and data segments do not overlap.

In the [default] reentrant mode all local "auto" variables are allocated and deallocated dynamically, i.e. they reside on a stack, necessary if recursive or interruptable functions are needed. This option generates relatively more and slower code than the static expanded mode. In the latter mode all function-level variables are put into static memory, except formal parameters which are always on stack.

The expanded modes enable 8051 family users to build very complex systems and use the full power of the C language. In the expanded memory models, C uses the internal RAM only for temporary storage (R0-R7, ACC, B, PSW, DPTR and 20 bytes of stack [at user specified locations].

Note that the C run-time system disables interrupts 10-20 cycles every time the virtual stackpointer (R6-R7) is updated. After such a sequence, the interrupt enable bit (IE.7) is restored to its original state.

The small and the expanded (medium and large) models have different C run time systems; thus modules generated by single-chip and expanded modes cannot be combined. If incompatible memory models are linked, the linker will flag either duplicate definitions or unresolved externals.

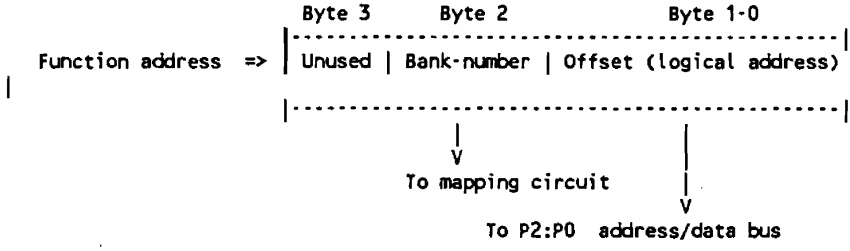
Modules from the large and medium modules may not be combined. However, modules from the large reentrant and large static models may be combined. Also, modules from the medium reentrant and medium static models may be combined. In the large or the medium memory models most code is typically written in the static mode combined with modules written in reentrant mode where recursive code and full support of interruptive code are required.

Review Appendix J for an overview of the memory organization in the expanded memory models.

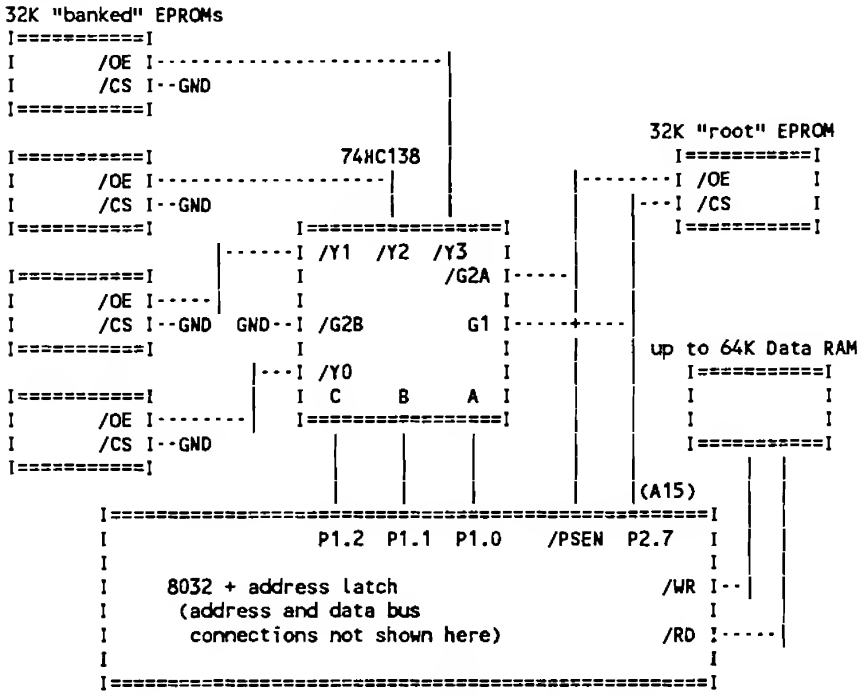
1.8 Banked Memory Models

The "banked" model is identical to the "large" model in terms of variable allocation and initialization (see previous section). However, the code area can be (transparently on the C level) extended with up to 256 blocks of memory. Blocksize can be up to 64K but the requirement to have a an always accessible "root" block, for practical reasons usually limit blocks to 16-32K.

To overcome the fact that function addresses are still only 16 bits wide, all non-local function calls are made *indirectly* through a 32-bit static function address. That is: A global function address does not point to the function, only to its indirect address location.This looks like:



To accomplish this, the compiler generates static function pointers (one for each non-local function definition in a module) and allocates them to a segment called FLIST. A typical PROM-based system could be configured like this:



The sample system utilizes four 32K banks to create a system with up to 160K of code. Port P1 was "sacrificed" to map the actual bank to be executed. Other ports may be used as shown in file **I18.s03** which contains the actual switching routines (user-configurable for other mapping hardware/schemes).

In the sample system the "root" EPROM is allocated to address 0-7FFF (hex) which is most practical since 8051 programs start at location zero after reset. The root memory contains all vital intrinsic library routines (i.e. support code like floating point arithmetic, rather than user-callable functions) needed by the C program. Due to the 8051 C run-time requirements all constant data must also be located in the root memory. The "banked" memory blocks of the sample system all start at logical address 8000 (hex) which means that the **-b** flag to the linker should be

set like this: **-bCODE=8000,8000,10000**. The first parameter is logical address of initial bank (P1 = 0) and the second parameter shows that 32K banks were used, while the third parameter (increment) gives bank-numbers 0, 1, 2, 3 etc. In the banked area there can only be C callable functions (both library type like **printf** as well as the user-written C routines).

IMPORTANT

1. No single module can be larger than the bank-size (over-sized modules result in linker error-messages).

2. It is recommended to keep module size considerably smaller than bank-size in order to avoid memory fragmentation (partially filled banks), as the linker only packs complete modules into banks.

3. The compiler will in the banked mode select the fast direct calling method (same as in the expanded mode) if a function is considered as local (i.e. has the storage-class **static** and is not referenced as a function pointer).

The banked model library resides in **cl8051b.r03**.

1.9 Register Usage

All compiler modes use registers R0-R7, ACC (A), B, PSW and DPTR, which is important to know when writing assembly language support routines or interrupt handlers. In the small (single-chip) mode registers are used as follows:

R0-R1,PSW, ACC,B,DPTR	Hold temporary results and do not require saving in assembly language routines.
R2-R3	Hold function return values for all types except "long" and "float" (structures are always returned as addresses). R2 contains the most significant part of the result for 16-bit results while R3 holds the entire value in case of "char" return values.

R2-R5	Hold function return values for "long" and "float" return values. R2 contains the most significant part of the result while R5 contains the least significant part.
R6-R7	May contain temporary results and must be preserved by assembly language routines.

In the expanded and banked modes registers are used as follows:

R0-R5,PSW, ACC,B,DPTR	As in small model.
R6-R7	Contain a 16-bit 'virtual' stackpointer which must not be manipulated at all by assembly language routines.

Note that R0-R7 can reside in any bank (selectable at link-time) but must always be in the same bank when C code is executed. That is, assembly language routines are free to switch banks at run-time but must always restore the orginal bank when returning control to C code. (Also, see how to specify register bank at link time in linker command file lnk8051.xlk).

1.10 Assembly Language Interface

Assembly language routines may be useful in optimizing time-critical sections of code.

The following sections describe the interface between the C system and assembly routines. Calling C routines from assembly is also possible but has few applications. For more details study the the compiler's output using the **-q** switch. Note that the banked-mode requires a considerably more complex handling and is not explicitly described in this document. The best way to create a bank-switched assembler routine is to create a C "shell" program that only declares the variables involved and also does simple accesses to the declared variables:

```
void func(int i, int j)
  {
    i;    /* Access to i */
    j;    /* Access to j */
  }
```

Then compile the program using the -A, -q and -L switches to get a pure assembly file (.s03) containing the proper function enter/return sequences as well as variable accesses. To this file user-written code can be added.

1.11 Small Model Parameter Stack

In the small [single-chip] model, parameters are pushed on the internal stack in reversed order with the RETurn address as top element.

The called function is responsible for 'popping' all variables from the stack before returning to the caller. If the called routine has a fixed number of parameters this is quite straight-forward, while a function with a varying number of parameters requires the contents of register R0 to restore SP before returning. **If only one single argument is passed R2-R5 contain that argument. If multiple arguments are passed, the second argument and up are on the stack.** Immediately after a function call the stack contains (internal RAM):

```
R2-R5 = Argument 1      [1, 2]
R1 contains the negated size of the parameter block
   (including any unpushed arguments)

 low memory
 -----------
| Argument n |
|     .      |
|     .      |
| Argument 3 |
| Argument 2 |          [1]
| Return addr| <--SP
 -----------
high memory
```

[1] Note that the first parameter resides in register R3 (8-bit), in registers R2-R3 (16-bit), or in registers R2-R5 (32-bit) with one important exception: "struct" and "union" objects are always pushed on the stack completely before a function call is performed.

[2] A special technique is used for "struct" and "union" return values. The caller reserves an area somewhere in its own "auto" space and gives the called function an address to that area as a first parameter (in R2-R3). The called function is in this case responsible for using that parameter as a return value location as well as setting registers R2-R3 to that address at the time of return.

The return value should be stored in R2-R5 if it is 32-bit, R2-R3 if it is 16-bit, and in R3 if it is 8-bit, when returning to the caller.

Now assume that a routine **unsigned long uadd (unsigned int a, unsigned int b)** is to be written. A routine that performs an unsigned addition and returns an unsigned long could in the small model look like this:

```
        NAME    sample_module
        PUBLIC  uadd
        EXTERN  _R              ; Bank [0,8,10H or 18H
        RSEG    CODE
uadd:   POP     _R+6            : Save return address
        POP     _R+7            ;     -"-
        POP     _R+4            ; Get MSbyte of b
        POP     ACC             ; Get LSByte of b
        PUSH    _R+7            ; Restore return address
        PUSH    _R+6            ;     -"-
        ADD     A,R3            ; Add LSBytes
        MOV     R5,A
        MOV     A,R2            ; Get MSByte of a
        ADDC    A,R4            ; MSByte
        MOV     R4,A
        MOV     R2,#0
        MOV     R3,#0
        JNC     $+3
        INC     R3
        RET
        END
```

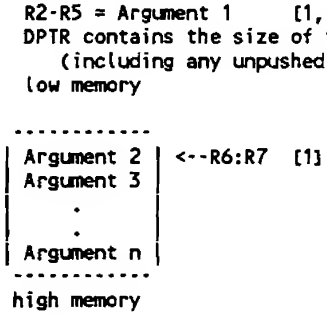
Note that when C variable items are addressed in the small memory model, and the address is not 'auto' (i.e. not the on stack), a check is made on the most significant byte of the address. If MSbyte is zero internal RAM access is assumed while not zero denotes access to the CODE memory space.

In the small memory model the 8051 has built-in pop instructions that are used to get parameters.

1.12 Expanded Mode Parameter Stack

In the expanded modes parameters are pushed on the virtual stack in reversed order while the RETurn address resides as the top element of the internal stack.

The called function routine is responsible for 'popping' all variables from the stack before returning to the caller. If the called routine has a fixed number of parameters this is straightforward (e.g. if the function is called with two int parameters it would pop four bytes from the stack total). However, a function being invoked with a varying number of parameters will be passed the size of the pushed parameter block in register DPTR. **If only one single argument is passed R2-R5 contain that argument. If multiple arguments are passed, the second argument and up are on the stack.** This is required so that registers R6, R7 are restored before returning. You should not modify R6, R7 as these contain a 16-bit 'virtual' stackpointer. Immediately after a function call the virtual stack contains the following (external RAM):



[1] Note that the first parameter resides in register R3 (8-bit), in registers R2-R3 (16-bit), or in registers R2-R5 (32-bit) with one important exception: "struct" and "union" objects are always pushed on the stack completely before a function call is performed.

[2] A special technique is used for "struct" and "union" return values. The caller reserves an area somewhere in its own "auto" space and gives the called function an address to that area as a first parameter (in R2-R3). The called function is in this case responsible for using that parameter as a return value location as well as setting registers R2-R3 to that address at the time of return.

The return value should be stored in R2-R5 if it is 32-bit, R2-R3 if it is 16-bit, and in R3 if it is 8-bit, when returning to the caller.

Note that the virtual stack grows from high to low addresses contrary to the internal stack that grows from low to high addresses.

Now assume that a routine **unsigned long uadd (unsigned int a, unsigned int b)** is to be written. A routine that performs an unsigned addition and returns an unsigned long could in the expanded mode look like this:

```
        NAME    sample_module
        EXTERN  ?POP_R4_R5_L17
        PUBLIC  uadd
        RSEG    CODE
uadd:   LCALL   ?POP_R4_R5_L17  ; Get b
        MOV     A,R3            ; Get LSByte of a
        ADD     A,R5            ; LSByte
        MOV     R5,A
        MOV     A,R2            ; Get MSByte of a
        ADDC    A,R4            ; MSByte
        MOV     R4,A
        MOV     R2,#0
        MOV     R3,#0
        JNC     $+3
        INC     R3
        RET
        END
```

Intrinsic library function calls to manipulate the virtual stack:

?POP_R3_L17	pop one byte into R3
?POP_R5_L17	pop one byte into R5
?POP_R2_P3_L17	pop two bytes into R2-R3
?POP_R4_P5_L17	pop two bytes into R4-R5
?POP_R2_R5_L17	pop four bytes into R2-R5
?PUSH_R3_L17	push R3 register
?PUSH_R5_L17	push R5 register
?PUSH_R2_P3_L17	push R2-R3 registers
?PUSH_R4_P5_L17	push R4-R5 registers
?PUSH_R2_R5_L17	push R2-R5 registers

?POP_R3_L17 pops for instance a 'char' data type (see section 1.4. Data Representation). **All these routines update R6-R7 and destroy the contents of ACC, DPTR, and R0-R1.**

1.13 Linking

Linkage of object modules created by the 8051 C-compiler requires several steps that are best carried out by using a linker command file (**xlink -f file**). To ease creation of such files a framework is supplied in the form of files **lnk8051s.xcl**, **lnk8051.xcl**, and **lnk8051b.xcl** for the small, expanded and banked modes respectively. The contents of **lnk8051.xcl** is shown below:

```
-!                -LNK8051.XCL-
88        XLINK command file to be used with the 8051 C-compiler V2.xx
        using the -m0 to -m3 options (expanded)
        Usage:  xlink  your_file(s)  -f lnk8051
        First: define CPU  -!
-c8051
-!  Assign/allocate a value for SP (change if not reg-bank #0) -!
-Z(IDATA)ISTACK=7
-!  Select register bank [0,8,10 or 18] -!
-D_R=0
-!  Setup all read-only segments (PROM). Usually at zero -!
-Z(CODE)CSTART,RCODE,CODE,CDATA,ZVECT,CONST,CSTR,CCSTR=0
-!  Setup all writable segments which must be mapped to external RAM. -!
-Z(XDATA)DATA,TEMP,IDATA,UDATA,ECSTR,WCSTR,XSTACK=0
-!  Load the 'C' library -!
cl8051
-z
-!  Code will now reside on file AOUT.A03 in format INTEL-STANDARD -!
```

The resulting code downloaded to an emulator, SDK-51 or PROM-programmer. Other formats than the default can be set with the -F option.

Note that the **cstartup** routine is always loaded automatically from the C library at link time. The resulting code (default in Intel-hex) is written on *output_file* and can be downloaded to an emulator, SDK-51 or PROM-burner.

The supplied linker command files contain several segments defined for the C run-time and the PROMming of code. Although your particular program might not be using all of these segments, you are recommended to leave these segment definitions in your linking file as you do not gain anything by removing a segment. The defined segments in the linker command file are (upper case is significant):

CSTART
: Used by **cstartup.s03** to make sure programs start a zero.

RCODE
: Used by routines called by the C code generator. (Banked mode: Also suitable for user-written assembler code that is not called from C (i.e. interrupt handlers and similar resident code).

CODE
: Used for C library routines and is also suitable for holding code for assembly language routines. Bank mode note: should only hold code which is callable from C.

FLIST
: Only generated in the banked mode.

CDATA,ZVECT,CSTR,CCSTR,CONST
: Are used for initializing C variables at startup and may not be used in a user program (see call to ?SEG_INT_L17 in **cstartup.s03**).

ISTACK
: Internal stack segment (size = stackspace).

DATA
: Used for allocating uninitialized variables. May also be used in assembly language routines.

UDATA
: Used for allocating variables that should be set to zero at startup. May also be used in assembly language routines.

UDATA,WCSTR,ECSTR
: Are used for initializing C variables at startup and may not be used in a user program (see call to ?SEG_INT_L17 in **cstartup.s03**).

XSTACK
: External stack segment (size = stackspace). Expanded and banked models only.

Other segment names than mentioned above may be used and can be placed anywhere in the applicable segment list (see A.10 linking), or mapped to a special area like an I/O device or battery-powered RAM. You can create your own code segments by using the compiler's -R option (see section 1.20). The best way to create your own unique data segments is to follow the procedure on page 1-23 and use the librarian to rename a data segement and then insert this new data segment in the linker command file.

1.14 Configuration Issues

If hardware must be initiated before the C program starts, code can be added to the file **cstartup.s03** that contains the initialization routine (in assembly source) that is supposed to be loaded at address zero. If it is not loaded at address zero control must still be transferred to the first address of the CSTART segment (only used in **cstartup.s03**). After modifications the **cstartup.s03** should be re-assembled and the corresponding object file **cstartup.r03** should replace the original startup routine. This is best accomplished by using the following sequence (here for the expanded library) with the library manager program (the small model library is cl8051s):

```
C>xlib
*def-cpu 8051
*rep-mod cstartup cl8051
*exit
```

For details on how to modify the output from **putchar** and **printf** please refer to file **putchar.s03**. The corresponding file for character input is also available in source format (**getchar.c**).

The **printf** and **sprintf** routines share a common formatter which is pre-installed in the library as module **_frm_wri**. This module contains a version of the formatter that does not support floating point numbers. It will complain about %e, %E or %f specifiers. If you need floating point formatting you should C-replace the original module with a (considerably larger) version which resides in file **frmwri.r03** (**frmwris.r03** small model and **frmwrib.r03** for banked model). Please note that you need to change the name of the file **frmwri*.r03** to **_frm_wri.r03** prior to replacing the common formatter. The replacement procedure is analoguous with the one **cstartup.r03**. The source for **printf.c**, **sprintf.c**, and **frmwri.c** are provided. These routines can easily be adapted to user-defined purposes such as creating a routine that writes formatted information to a non-standard display device. (Also see Appendix B, Libraries).

Since the "small" and "expanded" memory models are incompatible, the user-configurable as well as user-written assembly language routines must know what model to use. In order to ease things a bit the routines supplied in this package reference an include file **defmm.inc**. This file is supposed to be created by copying **defsc.inc** (small), **defem.inc** (expanded models) or **defmb.inc** (banked) to **defmm.inc**. Conditional assembly does the rest. Study **cstartup.s03**, **putchar.s03** and **getchar.c** for more info.

In some cases there is a need to keep variables unchanged even if power is turned off. In these systems some of the RAM is implemented in EEPROM or in a battery-powered memory device. To avoid the automatic variable initialization performed by the C system at startup (see file **cstartup.s03**) one of the following should be performed:

1. Remove the call to **?SEG_INIT_L17** (*never in the large or banked modes*). Disadvantage: All static or global C variables will contain undefined values at startup.

2. Declare the special memory contents in an assembly language file:

```
NAME      NON_VOLATILE_RAM
PUBLIC    last_sum
PUBLIC    user_id
ASEG                          ; Absolute mode selected
```

```
        ORG     1024            ; Assumed start address
last_sum DS     4               ; A "float"
user_id DS      11              ; char user_id[11];
        END
```

In the C modules you would then have to declare:

```
extern float last_sum;
extern char user_id[11];
```

It is also possible to make the variables relocatable using any segment name which is not in conflict with the C run-time system.

3. Declare the special memory contents in a, otherwise preferably empty C file (i.e. module) called for example **nvram.c**. Note: no initializers should be specified.

```
float last_sum;
char user_id[11];
```

Compile this file using the -P option to create declarations on a segment called UDATA:

```
C>c-8051 -P nvram
```

As the UDATA segment is normally cleared (set to zero) by **?SEG_INIT_L17** you must rename it (in this particular object file only) to something which is not in conflict with the C run-time system (e.g. NONVOLRAM):

```
C>xlib
*def-cpu 8051
*ren-seg nvram UDATA NONVOLRAM
*exit
C>
```

In the other C modules you would then have to declare:

```
extern float last_sum;
extern char user_id[11];
```

Now segment NONVOLRAM can be assigned to any applicable address with the -Z switch in XLINK.

1.15 Stack and Heap Size

By default the run-time stack size is set to 64 and 512 bytes for the single-chip and the expanded modes respectively. If these values need to be *increased* this can be done at link-time (before DUMP) by adding *+value* to the proper segment:

-ZISTACK+*value*=address	internal stack.
-ZSEGA,SEGB,XSTACK+*value*=address	external virtual stack.

If there is a need to change these values in the opposite direction this should be done by modifying the file **cstartup.s03** and replacing the original library module with the same sequence as described in the configuration section. The stack requires, as a general guideline, 2 bytes per level of subroutine nesting, 10 bytes for an integer operation, 20 bytes for a float operation, about 20 bytes at any time for C run-time library functions and at least 25 bytes for interrupt support.

The heap is only needed (and loaded) if the library routine **malloc** is called. The preset heap size is 2000 bytes. If this area must be changed, the file **heap.c** must be recompiled and re-installed in the C library again using XLIB.

1.16 Interrupt Routines

Interrupts are handled via user-defined low level routines written in C or assembler.

The IE (Interrupt Enable) and IP (Interrupt Priority) are the two critical interrupt registers controlling timing, interrupts and priorities. To interrupt the processor the appropriate IE bit has to be set high. (Note the 8051 responds to an interrupt after the current instruction being executed is finished. This can take up to 4 clock cycles). Interrupt sequence:

1. The CPU finishes current instruction in execution.
2. The PLC is stored on the stack.
3. All other interrupts of equal (or lower) priority are disabled.
4. A user-defined interrupt handler then stores CPU status information and starts processing the interrupt.
5. After the interrupt procedure is completed the CPU is returned to its pre-interrupt state.

Interrupt routines must be installed by modifying the file **cstartup.s03**. The segment CSTART must be loaded at address zero and is therefore the segment that should contain vectors (i.e. JMP's) to the appropriate interrupt handler. For interrupt vector locations consult the Intel Microcontroller Handbook. **Interrupt handlers written in C must have R0-R5 (expanded models), R0-R7 (small models), ACC, B, PSW and DPTR preserved before the call whereas assembly written routines only have to preserve registers actually used by the handler.**

If you intend to write interrupt routines in C while using the expanded memory models, the original module that manages the virtual stackpointer in the C library must be replaced. The reason for this is that updating the virtual stackpointer (R6:R7) is an "atomic" operation and must be performed with the interrupt system turned off (although only in the case where C interrupt routines are to be used). The interrupt version of the stackmanager resides in file **cinter.r03** (**cinterb.r03** for banked mode) and is inserted in the same way as **cstartup.r03** (see 1.13 configuration issues).

> *Please note that* **cinter.r03** *and* **cinterb.r03** *disables interrupts 10-20 cycles every time the virtual stackpointer (R6-R7) is updated. After such a sequence the interrupt enable bit (IE.7) is restored to its original state.*

Below is an example on how **cstartup.s03** can be modified to call a parameter-less C function "heater_overflow" as an interrupt handler:

```
        EXTERN  heater_overflow
; CSTART must be loaded at address
; zero to get this working!
        RSEG    CSTART
startup:
        SJMP    init
        ORG     startup+3
        SJMP    handler
          .
;-------------------------------------;
; Small memory model interrupt handler ;
;-------------------------------------;
```

```
handler:PUSH    ACC
        PUSH    B
        PUSH    PSW
        PUSH    DPH
        PUSH    DPL
        PUSH    _R+0
        PUSH    _R+1
        PUSH    _R+2
        PUSH    _R+3
        PUSH    _R+4
        PUSH    _R+5
        PUSH    _R+6
        PUSH    _R+7
        MOV     R1,#0  ; No parameters pushed
        LCALL   heater_overflow
        POP     _R+7
        POP     _R+6
        POP     _R+5
        POP     _R+4
        POP     _R+3
        POP     _R+2
        POP     _R+1
        POP     _R+0
        POP     DPL
        POP     DPH
        POP     PSW
        POP     B
        POP     ACC
        RETI

;---------------------------------------;
; Expanded memory model interrupt handler ;
;---------------------------------------;
handler:PUSH    ACC
        PUSH    B
        PUSH    PSW
        PUSH    DPH
        PUSH    DPL
        PUSH    _R+0
        PUSH    _R+1
        PUSH    _R+2
        PUSH    _R+3
        PUSH    _R+4
        PUSH    _R+5
        MOV     DPTR,#0  ; No parameters pushed
        LCALL   heater_overflow
        POP     _R+5
        POP     _R+4
        POP     _R+3
        POP     _R+2
        POP     _R+1
        POP     _R+0
        POP     DPL
        POP     DPH
        POP     PSW
        POP     B
        POP     ACC
        RETI
```

```
;--------------------------------------;
; Banked memory model interrupt handler ;
;--------------------------------------;
handler:PUSH    ACC
        PUSH    B
        PUSH    PSW
        PUSH    DPH
        PUSH    DPL
        PUSH    _R+0
        PUSH    _R+1
        PUSH    _R+2
        PUSH    _R+3
        PUSH    _R+4
        PUSH    _R+5
        MOV     DPTR,#heater_overflow
        LCALL   ?X_CALL_L18
        DW      0               ; No parameters pushed
        POP     _R+5
        POP     _R+4
        POP     _R+3
        POP     _R+2
        POP     _R+1
        POP     _R+0
        POP     DPL
        POP     DPH
        POP     PSW
        POP     B
        POP     ACC
        RETI
```

1.17 8051-Specific Functions and Libraries

Archimedes C-51 compiler has a useful set of special 8051 in-line functions to support bit manipulation and symbolic access to chip features (internal RAM/special function registers) directly in C (thus minimizing the need for assembly programming). **These in-line functions are only enabled when using the C-compiler -e option.**

The in-line functions can be used in the same contexts as ordinary functions with one exception: they cannot be referred to as addresses in initializers and in function pointer assignments. In addition they behave like *reserved words*.

io51.h is a header file that includes all the pre-defined byte and bit adresses for 8051/52/31/32 (**io515.h** supports Siemens 80515/535). **You must include the 'io51.h' header at the beginning of your file using the '#include <io51.h>' statement**

prior to using any of the special 8051 functions. You may add your own set of pre-defined byte and bit internal RAM/special function registers addresses (for any 8051 proliferation chips) by following the supplied framework in 'io51.h'. As an example, the byte-address for P0 is defined as:

```
#define P0          0x80
```

The (special function register) bit-address for P0.0 is defined below. (The "BIT" name-extension is included only to clearly separate bit addresses from other names. Underscore "_" is used in P0_0 as P0.0 is not an allowed C construct).

```
#define P0_0_BIT    0x80
```

The file **bio51.c** lists all the special 8051 in-line functions. The in-line functions conceptually work like the ANSI-type function declarations (casts performed automatically). (SFR = Special Function Register). Several of the in-line functions have one restricted address operand type called ICE which denotes that the operand must be an Integral Constant Expression rather than any type of integral expression.

The in-line functions conceptually work like the following ANSI-type (casts performed automatically) function declarations listed below. There are two functions available to access byte addresses and four functions available to support bit manipulation:

input

unsigned char input(ICE address);

Read any SFR (or internal RAM) byte address at location 0-255). Read value: 0-255.

output

void output(ICE address, unsigned char data);

Write to any SFR (or to internal RAM) byte address at location 0-255). Written value: 0-255.

read_bit

unsigned char read_bit(ICE bit_address);

Read any internal RAM/SFR bit address. Read value 0/1.

set_bit

void set_bit(ICE bit_address);

Set any internal RAM/SFR bit address.

clear_bit

void clear_bit(ICE bit_address);

Clear any internal RAM/SFR bit address.

read_bit_and_clear

unsigned char read_bit_and_clear(ICE bit_address);

Read and clear any internal RAM/SFR bit address using JBC. Read value: 0 or 1.

The following code (also available in file **bio51.c**) shows typical uses of the in-line functions:

```
#include "io51.h"              /* SFR definitions */

static char c;

void main(void)
  {
    output(P0,c);              /* Set P0.0 - P0.7 = c */
    set_bit(P0_1_bit);         /* Raise P0.1 */
    clear_bit(P0_1_bit);       /* And down with it again */
    if (! read_bit(P0_1_bit))
      c = input(P0);           /* Get that byte */
    set_bit(P0_1_bit);         /* Raise P0.1 */
    if (read_bit_and_clear(P0_1_bit))
      c = input(P0);           /* Get that byte */
  }
```

In addition to the functions described above there are two functions available to access external DATA memory and one to access CODE, which can be interesting if program variables are relatively few but there is still a need to manipulate a large buffer, control memory mapped I/O devices etc.

read_XDATA

unsigned char read_XDATA(unsigned int address);

Read any [external] DATA memory byte at location 0-65535. Read value: 0-255.

write_XDATA

void write_XDATA(unsigned int address, unsigned char data);

Write to any [external] DATA memory byte at location 0-65535. Written value: 0-255.

read_CODE

unsigned char read_CODE(unsigned int address);

Read any CODE memory byte at location 0-65535. Read value: 0-255.

The read_CODE in-line function is shown in the example below, which describes how to declare constant data elements that will be stored in ROM only - without taking up space in RAM when the large model is used. This can be useful when you have a large amount of constant data and a limited amount of RAM.

The read_CODE allows accessing this ROM-only data, a byte at a time, without having to resort to assembly code. For example, the following shows how to declare a C string that will be stored in PROM only:

```
/* file ROMDATA.C */
const char *hello = "Hello, world, this a ROM string\n";
```

Compile the file ROMDATA.C using the medium (-m2) memory model, which causes the initialized constant data to be placed into the PROM memory only (no RAM will be allocated). Below the program shows how read_CODE is used to access this string:

```
#include <stdio.h>
#define MAXLEN 80       /* set to size of longest string + 1 */

void main(void)
{
  extern const char *hello;
  char buf[MAXLEN];
  int c;

    /* copy the string byte-by-byte from PROM to buffer */
  for(c = 0; ((buf[c] = read_CODE((unsigned int)hello)) != 0);
      c++,hello++)
    ;
  printf("%s",buf);     /* print the string */
}
```

This program will function properly when compiled in the large model (-m0 or -m1). Please note that the -e compiler option must be used to enable the READ_code in-line function.

External RAM or external I/O devices may be accessed through a pointer variable as follows:

```
#define xaddr 0xA000   /* external device located at A000 hex */
.
{
 char port_val;

 *(char *)xaddr = 0x80;   /* writes 80 hex to A000 hex */
 port_val = *(char *)xaddr;  /* reads device at A000 hex */
}
```

Observe that pointers are not integers, i.e. the C-compiler is very strict in typechecking. This is especially crucial to look out for if you are porting 'generic C-code' which you have used with an older type of a compiler, which may have allowed integers to be freely assigned to pointers (and vice-versa) without complaining. (See Kernighan & Ritchie section 5.6 Pointers are not Integers). If you must assign a value to a pointer variable, "cast" the integer to a pointer type as shown in the example above.

The Archimedes C-compiler kit includes most of the important C-library functions that apply to microcontrollers. The libraries for the large and medium memory models are in the **cl8051.r03** file. The corresponding small memory model c-library functions are in **cl8051s.r03** file. **Review Appendix B: Libraries, prior to using any of the C library functions.**

1.18 Operating Instructions

C-51 [*options*] *sourcefile* [*options*]

The Archimedes C-compiler is a single executable file that is invoked by issuing the name of the compiler program, **"c-51"**, followed by a source filename with the default extension .c. The source file can optionally be preceded and/or succeeded by compiler option switches recognized by a hyphen character (-). Source filename and compiler switches must be separated by tabs or spaces only. Additional command line input can also be given through a target dependent 'environment' variable and for very complex invocations through a command file (with the -f switch). The latter offsets the current 128 character limit imposed by MS-DOS. If C-51 is invoked without any options given, directly followed by a <CR>, the compiler lists all the options available (=help command).

1.19 Files

The C-compiler processes one source file at each invocation and the generated code is written on an object file which by default has the basename of the source file plus the 8051-specific (.r03) extension. The basename is the name of a file without directory information and filename extension (i.e. the basename of **\usr\myfile.c** is **myfile**). The 8051-dependent filename extension on object files is listed by invoking the compiler without any parameters. If no object file is wanted, specify: **-o nul** which forces the compiler to write object code on the MS-DOS 'null' file.

Filenames can be given in any mixture of upper and lowercase letters while command line switches (options) are interpreted "as is". Default file extensions can always be overridden if needed (i.e **myfile.g** specified as source file will force the compiler to open **myfile.g** while **myfile** will be treated as **myfile.c**). In addition to the object file, the compiler can optionally produce a list file (**-L** or **-l** switches) and/or a file with symbolic assembly code that can be processed by the assembler (**-A** or **-a** switches). No object file will be generated if errors are detected during compilation. However, if only warnings ocurr, an object file will be created.

1.20 Compiler Switches and Options

The compilation process can be controlled by an extensive set of command line switches/options. The position of the options in the command line is of no importance except for **-I** commands. Note that all switch commands with a **file** argument must have the filename separated from the switch flag with tabs or spaces. All other type of arguments should start immediately after the switch character. The table below summarizes all the options. This summary is also presented on the screen by typing "C-51" followed by <CR>. All options are described in more detail at the following pages. CASE IS SIGNIFICANT IN COMPILER SWITCH OPTIONS.

```
Usage:          c-51 (<options>) <sourcefile> (<options>)
Sourcefile:     'C' source file with default extension: .c
Environment:    QCC_51
Options (specified order is of no importance):
-V              Verbose error-messages
-o file         Put object  on: <file> <.r03>
-Oprefix        Put object on: <prefix> <source> <.r03>
-b              Make object a library module
-P              Generate PROMable code
-g(OA)          Enable global typecheck
                   O: Disable object code type-info
                   A: Depreciate K&R-style functions
-m0             Select memory model: large reentrant (default)
-m1             Select memory model: large static
-m2             Select memory model: medium reentrant
-m3             Select memory model: medium static
-m4             Select memory model: small reentrant
-m5             Select memory model: small static
-m6             Select memory model: banked reentrant
-m7             Select memory model: banked static
-w              Disable warnings
-s              Optimize for speed
-z              Optimize for size
-j              Put static and local symbols in the object
-y              Put "strings" into variable section
-c              Make plain 'char' = 'signed char'
-e              Enable processor dependent extensions
-f file         Extend command line with <file> <.xcl>
-r              Generate calls to run-time debugger
-l file         Generate a list on: <file> <.lst>
-Lprefix        Generate a list on: <prefix> <source> <.lst>
-tn             Set tab spacing between 2 and 9 (default 8)
-x(DFT2)        Generate cross-reference list
                   D: Show all #defines
                   F: Show all functions
                   T: Show all typedefs
                   2: Dual line space listing
-q              Put mnemonics in the list
-T              List 'active' lines only (#if etc. true)
-i              List #included files
-pnn            Page listing with 'nn' lines/page (10-150)
-F              Generate formfeed after each listed function
-a file         Generate ASM on: <file> <.s03>
-Aprefix        Generate ASM on: <prefix> <source> <.s03>
-Hname          Set object module header = 'name'
-Rname          Set code segment = 'name'
-DSYMB          Equivalent to: #define SYMB 1
-DSYMB=xx       Equivalent to: #define SYMB xx
-USYMB          Equivalent to: #undef SYMB
-Iprefix        Add #include search prefix
-G              Open standard input as source
-S              Silent operation of compiler
-C              Enable nested comments
```

-V Verbose error-messages

By default the compiler will generate diagnostic messages in the form:

"file",line_no Error[*err_no*]: *Diagnostic*
or
"file",line_no Warning[*warn_no*]: *Diagnostic*

where *"file"* is the current source or include filename, *line_no* the local line number in that file and *err_no/warn_no* is the error or warning number respectively. By giving the -V switch the erroneous line and a pointer to the faulty spot will also be displayed:

```
      if (i)) j++;
----------------^
  "main.c",870 Error[110]: ')' unexpected
```

-o file Put object on: <file> <.r03>
-Oprefix Put object on: <prefix> <source> <.r03>

By default the compiler will generate object code on a file which name is the basename of the invoking source file plus the 8051-specific extension (.r03). With the **-o** switch you can force the compiler to generate code on any file.

Sometimes it is convenient to have source files on one directory, object on another directory and compile from a third directory. In these cases the **-Oprefix** can be used to redirect object files from the default (current) directory where they usually are created. Note that no special interpretation of the prefix argument is performed. It is just inserted at the front of the default object filename. The Prefix would typically be a pathname. You may not mix **-o** and **-O** in a C command line. Example:

```
C-51 test.c -O\upper  (output file on \upper\test.r03)
```

-b Make object a library module

By default the compiler generates an object module with the *program* attribute (see assembler section 2.14). With the -b switch the object modules will get *library* attribute.

-P Generate PROMable code

If a C program contains statically initialized data or depends on static data without explicit initializers set to zero, it cannot be put into read-only storage directly since variable objects must be mapped to RAM which needs to be initialized at power-up time. With the **-P** command line switch static initializers will be put into its own memory segment which will be mapped into ROM address space and at start-up copied into RAM by the C run-time system. If the recommended linking scheme is used, all operations and memory allocations will be automatically and transparently performed.

-g{OA} Enable global typecheck

With the **-g** switch the compiler enters a mode which is more strict (i.e warns) about the use of certain C constructs like:

- Calling undeclared functions
- Undeclared K&R formal parameters
- Missing "return" values in non-void functions
- Unreferenced local or formal parameters
- Unreferenced "goto" labels
- Unreachable code
- "string" or 'c' containing non-printable characters

-g cont.

- Unmatching or varying parameters to K&R functions
- #undef on unknown symbols
- Valid but ambiguous initializers
- Constant array indexing out of range
- Old-Style K&R-functions [_gA only]

In addition to the checks performed at compile-time the -g switch enables the generation of linker type information which is used for module interface checking. The -g command does not generate more code but takes additional time to process in the compiler and in XLINK.

The -g switch can optionally be succeeded by the letter O and/or A. O implies that no object code information is to be generated but full type-check should still be performed *within* the module. The A modifier forces the compiler to complain (i.e warn) about the pre-ANSI type of function declarators that are in this document referred to as K&R.

Note that modules compiled without the **-g** *option are considered as typeless and global declarations in such modules will match (in XLINK) any declarations in other modules regardless if these modules where compiled with or without the* **-g** *option.*

-w Disable warnings

If the -w flag is given in the command line, warning messages will not be displayed although they will still be counted and will show up in compilation statistics.

-mx Memory model selection. See sections 1.5-7.

-s	Optimize for speed
-z	Optimize for size

By default the compiler performs code-optimizations, however, the level and goal of the optimization can be further controlled by the -s and -z switches. Only one of these flags can be activated during a single compilation.

-j Put static and local symbols in the object

For debugging purposes the compiler can optionally insert internal labels and static symbols in the object code.

-y Put "strings" into variable section

By default C string literals are assumed to be read-only. With the -y switch activated strings will be generated as initialized variables. Arrays initialized with strings (i.e. char c[] = "string") however, are always treated as ordinary initialized variables.

-c Make plain 'char' = 'signed char'

Makes all character data types signed by default (for compatibility with some other C-compilers).

-e Enable target dependent extension. See 1.16.

-r Calls to run-time debugger (future use).

-l file Generate a list on: <file> <.lst>
-Lprefix Generate a list on: <prefix> <source> <.lst>

Use one of these commands to generate a C list file. Filenames are treated analoguous with the -o and -O switches.

-tn Set tab spacing between 2 and 9 (default 8)

In C list files tabulators are converted to space with the value specified with the -t switch.

-x{DFT2} Generate cross-reference list

When the -x switch is given a list of all global symbols and their meaning will be printed after the C source code. By default the -x switch without any arguments will show all variable objects and all *referenced* functions, #defines, enums and typedefs. If the -x switch is succeeded by one or more of the arguments **D,F,T** and 2 the list will be expanded to also include:

D: Unreferenced #define symbols

F: Unreferenced function declarations

T: Unreferenced enum constants and typedefs

2: Dual line spacing between symbol entries

-q Put mnemonics in the list

Merges C source code with generated native code in symbolic assembly format on the list file. For debugging purposes.

-T List 'active' lines only (#if etc. true)

-i List #included files

-pnn Page listing with 'nn' lines/page (10-150)

-F Generate formfeed after each listed function

-a file Generate ASM on: <file> <.s03>
-Aprefix Generate ASM on: <prefix> <source> <.s03>

When one of these switches is activated the compiler will generate symbolic assembly code on either a specified file or use the prefix argument combined with the basename of the source file as the target file. The generated code can then be assembled by the Archimedes assembler. <.s03> is an 8051-specific extension.

-Hname Set object module header = 'name'

Occassionally it is impractical to have the same name on the object module as the base-name of the source file. A typical example is when the compiler is driven by some other pre-compiler that typically always uses the same output file. XLINK would then complain about 'duplicate modules' if several modules were linked. With the **-H** switch the default name can be *overridden.*

-Rname Set code segment = 'name'

Executable code is usually put on a segment named CODE. In the cases were a module must be loaded at a very special address the **-R** switch makes it possible to generate a unique segment name which can be placed at any address by XLINK. This is useful in bankswitched or in systems using multiple separate PROMs as it facilitates the setting up of different segments in the linker file.

-Iprefix Add #include search prefix

To force the include command to search at other directories for a specified file, any number of -I options with directory information can be issued. Example:

-I\usr\proj

Note that the prefix parameter is only added to the include file name without any interpretation (i.e. **-I\usr\proj** would make the compiler try to open **\usr\projinclude_file** rather than **\usr\proj\include_file**). Also see section 1.20 about Include Files.

-f file Extend command line with <file> <.xcl>

The command line to the C-compiler can be extended to almost any length by using a command file which should contain command line parameters formatted in the same fashion as the ordinary command line. Note that in command files, the newline (<CR>) character is considered equivalent to tabs and spaces (i.e end of file is the end of the command file) which makes command file data more readable on printers etc. The default file extension on command files is **.xcl**. To make it possible to #define string literals etc. double quotes may be used around space separated items to make them appear as one:

```
"-DEXPR=F + g"              #define EXPR F + g
"-DSTRING=""micro proc"""   #define STRING "micro proc"
```

-DSYMB Equivalent to: #define SYMB 1
-DSYMB=xx Equivalent to: #define SYMB xx

With the -D switch you can do a command line #define of a symbol. This is particularly suited for configuration purposes or in conjunction with conditional compilation to disable or enable calls to user-written trace routines. Any number of symbols can be defined although you may also have to use the -F switch to get the command line length required.

-USYMB Equivalent to: #undef SYMB

When the compiler is invoked a number of pre-defined #define symbols exist. If any of those symbols is in conflict with (= already used) a user #define symbol, that symbols intial definition can be turned off with the -U switch. The pre-defined symbols are:

Archimedes_C	
__FILE__	"current source file name"
__LINE__	"current source line number"
__TIME__	"hh:mm:ss"
__DATE__	"Mmm dd yyyy"

-G Open standard input as source

With the -G option switch a C program can be read directly from the keyboard. The source (dummy) filename is set to **stdin.c**.

-S Silent operation of compiler

Makes compiler turn off sign-on message and compilation statistics report.

-C Enable nested comments

When -C is activated comments will nest to any level. Without the -C command line option, the compiler will warn about but ignore nested comments. The purpose of the -C flag is to make it easy to "comment out" parts of a C program that could also contain comments.

Note that the extended command-line data for setting defaults etc. can be given through a target-dependent environment variable:

set *ENVIRONMENT_VARIABLE*=**-V -s -DCONFIG=43**

The name of the actual environment variable can be found by invoking the compiler without paramets (=help command). Then the name will be displayed as:

Environment: *ENVIRONMENT_VARIABLE.*

1.21 Include Files

Include files are typically used to keep common definitions available to the different modules forming a program. The include files may be specified two ways: Within angle brackets (**<file>**) or as a C string literal (**"file"**).

When an #include directive is found the compiler tries to open the specified file in the following order:

1. Only for **"file"**: File prefixed with the directory part of the invoking source file. That is, if the invoking file name is **..\src\sieve.c** the include file **"std.h"** will force the compiler to try to open the file **..\src\std.h**.

2. File name prefixed with the arguments of any optional **-I** switch commands given.

3. File name prefixed with arguments of an optionally defined environment variable named C_INCLUDE. Note that multiple search paths can be specified in C_INCLUDE by separating arguments with semicolon like:

set C_INCLUDE=\usr\proj\;\headers

4. Lastly the file is opened using the 'naked' file name.

Note that arguments to -I switches and/or to C_INCLUDE are just added at the front of the #include file name without any interpretation. That is, a directory prefix must also contain the terminating backslash.

If the compiler fails to open a file using the steps above the compiler aborts the current compilation.

1.22 Compiler Diagnostics

The error messages produced by the C-compiler falls into six categories:

1. Command line errors
2. Compilation warning messages
3. Compilation error messages
4. Compilation fatal error messages
5. Memory overflow message
6. Compiler internal errors

Command line errors

Command line errors occur when the compiler is invoked with bad parameters. The most common situation is that a file can not be opened. The command line interpreter is very strict about duplicate, misspelled or missing command line switches. However, it produces error messages that point out the problem in detail.

Compilation warning messages

Compilation warning messages are produced when the compiler has found a construct which is typically due to a programming error or omission. Appendix D lists all warning messages. (Also, see section 1.20 regarding the usage of the -V switch).

Compilation error messages

Compilation error messages are produced when the compiler has found a construct which clearly violates C language rules. Note that the Archimedes C-compiler is more strict on compatibility issues than many other C-compilers. In particular pointers and integers are considered as incompatible when not explicitly casted. Appendix D lists all error messages. (Also, see section 1.20 regarding the usage of the -V switch.)

Compilation fatal errors

Compilation fatal error messages are produced when the compiler finds a user error so severe that further processing is not meaningful. After the diagnostic message has been issued the compilation is immediately terminated. Appendix D lists all compilation error messages (some marked as fatal). (Also, see section 1.19 regarding the usage of the -V switch).

Memory overflow message

The Archimedes C-compiler is a memory-based compiler which in case of a small host system memory or very large source files may run out of memory. This is recognized by a special message:

```
***COMPILER OUT OF MEMORY***

Dynamic memory used: nnnnnn bytes
```

If such a situation occurs, the cure is either to add system memory or to split source files into smaller modules. However, with 512K RAM the compiler capacity is sufficient for all reasonably sized source files. If memory is on the limit, please note that the **-q -x** and **-P** command line switches cause the compiler to use more memory.

Compiler internal errors

During compilation a number of internal consistency checks are performed. If any of these checks fail, the compiler will terminate after giving a short description of the problem. Such errors should normally not occur and should be reported to Archimedes technical support group.

1.23 C-51 Compatibility.

The Archimedes C-51 implements the full ANSI standard C language and the most important library functions for microcontroller development. This chapter gives a short description of the most significant differences of the proposed ANSI standard versus the K&R definition. (Also, review Appendix B: Libraries for an overview of the library functions).

Keywords

Deleted keyword: **entry**

New keywords:

const An attribute that tells that a declared object is non-modifiable:

```
const int i;      /* constant int */
const int *ip;    /* variable pointer to constant int */
int *const *ip;   /* constant pointer to variable int */


const int i;      /* constant int */
const int *ip;    /* variable pointer to constant int */
int *const ip;    /* constant pointer to variable int */

typedef struct
  {
    char *command;
    void (*function)(void);
  } cmd_entry;

const cmd_entry table[]=
  {
    "help",     do_help,
    "reset",    do_reset,
    "quit",     do_quit
  };
```

volatile An attribute with same syntax as const but tells that the value in the object may be modified by hardware and should not be "optimized out".

signed Analoguous to unsigned. Can be used before any integral type-specifier.

void Is a type-specifier that can be used to declare function return values, function parameters and "generic" pointers.

```
void f();       /* A function without return value */
type_spec f(void);  /* Function with no parameters */
void *p;     /* Generic pointer.  Can be casted to any
                other pointer and is assignment compa-
                tible with any pointer type */
```

enum Enumeration constants is a way to generate sequential integer constants that can replace #defines in some contexts.

```
enum {zero,one,two,step=6,seven,eight};
```

Data types

The complete set of basic data types now includes:

```
{unsigned | signed} char
{unsigned | signed} int
{unsigned | signed} short
{unsigned | signed} long
float
double
long double
*          /* Pointer */
```

Observe that pointers are not integers, i.e. the C-compiler is very strict in typechecking. This is especially crucial to look out for if you are porting 'generic C-code' which you have used with an older type of a compiler, which may have allowed integers to be freely assigned to pointers (and vice-versa) without complaining. (See Kernighan & Ritchie section 5.6 Pointers are not Integers). If you must assign a value to a pointer variable, "cast" the integer to a pointer type, e.g. num_ptr = (int *);. (Also, see example in section 1.17).

As in any standard C, integer or character hex constants may be entered as in the "0xA800" format.

"Signed char" or "schar" is an ANSI-standard 1-byte element. It can be used for both characters and numbers range (-127 to +127).

Register variables as a storage class is recognized but currently ignored due to the register requirements of the C run-time.

Function prototypes

Function prototyping is a way to specify parameters in function declarations as well as in function definitions that has been added by ANSI. Prototyping results in safer and under some circumstances more compact programs. The following example shows the difference between K&R declarations and prototypes:

K&R	Prototype
`extern int func();`	`extern int func(long val);`

`abort(s)`
`char *s;`
`  {`
`  }`
	`void abort(char *s)`
	`  {`
	`  }`

Now if "func" is called with an "int" parameter the K&R version will fail to work if "int" has a different size than "long" while the prototyped version vill extend the "int" argument in the same way as an assignment operator would. When prototyped functions are used, arguments are always checked for compatibility with the declaration, whereas K&R type of declarations are only checked when the **-g** option is on. The prototyped format does also specify how to denote a variable number of arguments. This is done by writing three dots in the place of a formal parameter. Also see library function "printf".

If external or forward references to prototyped functions are used, a prototype declaration should appear before the call.

Hexadecimal numbers in strings

In addition to the original octal constants (\nnn), ANSI allows hexadecimal constants (one to three digits) by using a backslash followed by x and the hexadecimal digits.

```
#define  Escape_C  "\x1bC"
```

Note that the zero was needed to avoid that the letter "C" was interpreted as a part of the hexa decimal constant.

Structure and union assignments

The current ANSI draft allows functions and the assignment operator to have arguments that are "struct" or "union" type rather than only pointers to such objects. Functions may also return structures or unions.

```
struct s a,b;  /* struct s declared earlier */
struct s f(struct s parm);

    a = f(b);
```

Linkage Model

There are some differences between C compilers currently on the market when *variable objects* are to be shared among modules (= files). For *functions* most C compilers perform identically. In Archimedes's portable C compiler, the scheme recommended in the ANSI supplementary document *Rationale For C* is used. This linkage model is called "Strict REF/DEF" and requires that all modules execept one use the keyword *extern* before the variable declaration. Example:

Module #1	Module #2	Module #3
`int i;`	`extern int i;`	`extern int i;`
`int j=4;`	`extern int j;`	`extern int j;`

New Pre-processor Directives

#elif *expression*
Any C lines

is equivalent to:

#else
#if *expression*
Any C lines
#endif

#error *any_valid_C_tokens*

It is a directive which is supposed to be used in conjunction with conditional compilation. When the #error directive is found the compiler stops with an error message (if compilation was not turned off by earlier #ifdef's etc).

1.24 C-Compiler Extensions.

To better support generation of very target-dependent code while still using a single source file for more than one target, a number of small C language extensions has been added to the Archimedes C-Compilers. For applications, study the #include file stdarg.h.

1. The macro **___TID___** returns an integer constant with the following coding:

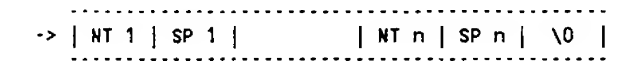
```
15            8 7          4 3          0
-----------------------------------------
| Target_IDENT | -v option | -m option |
-----------------------------------------
```

Target_IDENT:	A unique number for each target (execute `printf("%d",__TID__>>8)` to get the value)
-v option:	If compiler has a -v option this field holds the value (otherwise is 0)
-m option:	If compiler has a -m option this field holds the value (otherwise is 0)

2. The unary operator **_argt$** has the same syntax and argument as **sizeof** but returns a normalized value describing the *type* of the argument according to the following:

unsigned char	1
char	2
unsigned short	3
short	4
unsigned int	5
int	6
unsigned long	7
long	8
float	9
double	10
long double	11
pointer/address	12
union	13
struct	14

3. The reserved word **_args$** returns a *char array* (char *) that contains a list of elements describing the formal parameters of the currently compiled function (i.e. **_args$** can only be referred to inside function definitions). Parameter list layout:



Description: NT holds a Normalized Type (same values as for **_argt$**) whereas SP holds the Size of a Parameter (although parameters in excess of 127 bytes will only show up as 127 on the list).

The list ends when a type field with a zero value has been found (no more parameters or a "..." declaration).

4. The character $ has been added to the set of valid characters in identifiers to maintain compatibility with DEC/VMS C.

5. The ANSI-specified restriction that the **sizeof** operator cannot be used in #if and #elif expressions has been eliminated.

1.25 Sample Code

Review Appendix I for a larger code example.

The short program below illustrates the use of the Archimedes C-51 cross-compiler with the **-L** and **-q** compiler switches initialized. The **-L** switch generates a list file. The **-q** switch puts mnemonics in the list, i.e. merges C source code with generated native code in symbolic assembly format in the list file, which is useful for debugging purposes.

Source [code.c]:

```
char c[]="Some good old ASCII";
int i=78;
int j;

main()
{
  float res;
  static int a=4;
  i += j;
  exit();
}
```

Invocation:

```
C-51 code.c -q -L
```

Output:

```
#########################################################################
#                                                                       #
#     Archimedes 8051/2 C-Compiler V1.00A/MD2   20/Jan/86  12:10:56     #
#                                                                       #
#           Memory model  =  reentrant mode                             #
#           Source file   =  code.c                                     #
#           List file     =  code.lst                                   #
#           Object file   =  code.r03                                   #
#           Command line  =  code.c -q -L                               #
#                                                                       #
#                          (c) Copyright Archimedes Software Inc. 1986 #
#########################################################################

\   0000                    NAME    code(16)
\   0000                    RSEG    CODE(0)
\   0000                    RSEG    DATA(0)
\   0000                    PUBLIC  c
\   0000                    EXTERN  exit
\   0000                    PUBLIC  main
\   0000                    PUBLIC  i
\   0000                    PUBLIC  j
\   0000                    EXTERN  ?ENTER_L008
\   0000                    EXTERN  ?SI_ADD_ASG_L004
\   0000                    EXTERN  ?LEAVE_L008
\   0000                    RSEG    CODE
    1           char c[]="Some good old ASCII";
    2           int i=78;
    3           int j;
    4
    5           main()
    6           {
\   0000              main:
\   0000  7A00              MOV     R2,#0
\   0002  7B04              MOV     R3,#4
\   0004  120000            LCALL   ?ENTER_L008
    7             float res;
    8             static int a=4;
    9             i += j;
```

```
\   0007  900016            MOV     DPTR,#j
\   000A  E0                MOVX    A,@DPTR
\   000B  A3                INC     DPTR
\   000C  FC                MOV     R4,A
\   000D  E0                MOVX    A,@DPTR
\   000E  FD                MOV     R5,A
\   000F  7B14              MOV     R3,#LOW(i)
\   0011  7A00              MOV     R2,#HIGH(i)
\   0013  120000            LCALL   ?SI_ADD_ASG_L004
  10            exit();
\   0016  900000            MOV     DPTR,#0
\   0019  120000            LCALL   exit
  11          }
\   001C  020000            LJMP    ?LEAVE_L008
\   0000                    RSEG    DATA
\   0000              c:
\   0000  536F6D65          DB      'Some good old ASCII',0
\   0004  20676F6F
\   0008  64206F6C
\   000C  64204153
\   0010  43494900
\   0014              i:
\   0014  004E              DW      78
\   0016              j:
\   0016  0000              DB      0,0
\   0018              ?0000:
\   0018  0004              DW      4
\   001A                    END

Errors: none
Warnings: none
Code size: 31
```

2.1 Assembler - Introduction

The Archimedes assembler is a powerful relocating MACRO assembler with a very high degree of compatibility with other assemblers. Key design objectives have been ease of use, support for modularized programming and a good interface to high level languages. Assembly error messages are listed in Appendix E.

2.2 General Characteristics

The two-pass assembler executes as a single program. No temporary files are created during execution. The assembler and the linker use an internal 32-bit structure for all calculations which makes it possible to generate virtually any amount of code.

The following table shows the compatibility with most other 8051 assemblers. **For more detailed information regarding compatibility with the Intel 8051 assembler, please review section 2.22 Intel ASM Compatibililty.**

Compatible

- Machine instructions (names and syntax)
- Constant defining directives (DB and DW)
- Space allocation directives (DS)
- Delimiters
- Labels
- Constants
- The basic operators (+,-,*,/)
- ORG:s and EQU:s
- Absolute output format (generated by XLINK)

Not compatible

- Relocation directives
- Extended operators
- Conditional assembly
- Assembler options and controls
- Macro processing

Note that even the items stated as "not compatible", in many cases only have a different syntax compared to the original assembler definition (i.e. the same functions can readily be achieved).

You need to have the microcontroller manufacturer's documentation detailing the chip as well as summarizing the assembler commands. (The Intel 8051 Microcontroller User's Manual is required and the MCS-51 Macro Assembler User's Guide is valuable). These manuals have complete instruction sets not included in this manual.

In all examples user input is underlined.

2.3 Operating Instructions

```
A8051 <Source file> [<List file>] [<Object file>] [<Options>]
```

After the assembler program name (A8051), a <Source file> should be specified. There are two default file extensions on source files, a primary and a secondary. First the assembler tries to open a file with the primary extension, MSA, and if this fails, it will try with a secondary, 8051 extension (S03).

If no <List file> is specified the assembler assumes that no list is wanted and no list is generated. The default file extension on list files is LST.

The <Object file> name is by default equal to that of the source file with an extension of .R03. If no object file is wanted, specify the system "bit bucket" (NUL).

In all file specifiers the default file extension can be overridden. File names and options can be written in any mix of upper and lowercase letters.

For details on the <Options> parameter, refer to section 2.4.

All parameters should be separated by spaces or commas, and if a default parameter is wanted, it can be specified by writing an empty parameter delimited by commas (',,').

If no <Source file> is given, the assembler assumes a mode where no implicit default values are inserted. That is, some kind of response must be issued for all parameters. In this mode prompters are displayed that tell what type of parameter is asked for. This is the most practical mode to use if you have forgotten the start sequence. A <CR> in response to a prompt, means the default value will be used. Some examples of valid assembler invocation commands:

```
a8051 myfile

        <Source file>:          MYFILE.MSA (or .S03)
        <list file>:            No list generated
        <Object file>:          MYFILE.R03 (default)
        <Options>:              No options specified

a8051 myfile,,obj

        <Source file>:          MYFILE.MSA (or s03)
        <List file>:            No list generated
        <Object file>:          OBJ.R03
        <Options>:              No options specified

a8051 myfile.tst listing

        <Source file>:          MYFILE.TST
        <List file>:            LISTING.LST
        <Object file>:          MYFILE.R03 (default)
        <Options>:              No options specified

a8051 myfile prn:,,x p=41 s

        <Source file>:          MYFILE.MSA (or .s03)
        <List file>:            PRN
        <Object file>:          MYFILE.R03 (default)
        <Options>:              X, P=41 and S
```

```
a8051                                   No source file => prompt!

    Archimedes A8051 Assembler V1.80/MD2
 (c) Copyright Archimedes Software, Inc. 1985

Source file [.msa/.s03]=myfile          MYFILE.S03 (or MSA)
List file [.lst]=prn:                   On the printer please
Object file [myfile.r03]=<CR>           = default = MYFILE.R03
Options=?                               Get today's menu
Options available:
   X     Cross reference and symbol table
   P=nn  Paged list with <nn> lines/page
   E     List only errors
   F     Formatted list
   W     Wide list with file nest level
   S     Put local symbols in object
Options=s x p=41                        S, X and P=41
```

2.4 Options

The last parameter in the assembler invocation command contains the assembler options. The available options are:

X A cross reference and symbol table is produced. The X option is equivalent to the LSTXRF directive. Default is no table output.

P=nn Paged list with <nn> lines/page, where 10 <= nn and nn <= 150. The P option is equivalent to the PAGSIZ <nn> and LSTPAG+ directives. Default is a linear (non-paged) list.

E List only the erroneous lines on the <List file>. This option can save you lot of printer paper, time and disk space!

F Format the assembly list into fixed fields, regardless of how input is formatted. Equivalent to the LSTFOR+ directive. Default is that input is listed "as is", but with tabulators expanded.

W List <total lines><nest level><local line> Default is that only the total number of source lines read is listed. The W option is equivalent to the LSTWID+ directive.

S Put all symbols in the object code. The local as well as the global symbols will show up in XLINK load maps, with their values properly relocated. The S option is equivalent to the LOCSYM+ directive. Default is that only PUBLIC symbols are put in the object code.

? Give a menu of the available options and then ask the user to specify the options wanted.

2.5 Error Messages

All errors are issued as complete messages, that is, the messages should be self-explanatory. A typical example:

```
Error in 349: Undefined label: IO_op
        lcall     IO_op
                  ^
```

First a line with the diagnostic (with source line number) is written, then the erroneous source line, and last, a line with a pointer to the faulty spot. If include files are used, error messages will be preceded by the total source line numbers read, and also by the source line number and name of *current* file:

```
Error in 296/43 in "globdef.s03" : Invalid instruction
        dv    5       Reserve parameter space
          ^
```

Error messages will always be printed on the terminal, as well as on the (optional) list file. Also study section 2.4, concerning the 'E' option. Error messages are as few as possible and errors are reported as soon as possible. This means that if an error is flagged in the first pass, it will in most cases not be reported in the second pass.

In addition to assembler error messages, you may also get Pascal (the implementation language) I/O errors. These messages are normally easy to interpret. Note that XLINK will abort linkage of modules containing assembly errors!

Please refer to Appendix E for a complete listing of assembly error messages.

2.6 Output Formats

The assembler's output, i.e. the object code, is meant to be processed by the Archimedes Linker (XLINK). See chapter 3.11 for an overview of the Linker's output format.

The output from the Linker (XLINK), is in absolute format (see section 3.11) typically compatible with other vendor's debug programs, emulators and PROM programmers.

2.7 List Format

Assembly list information is put into four fields: Source line number, address field, data field and last, the source line. For more information, also read about the LSTWID, LSTEXP, LSTFOR, and LSTXRF directives. The address and data fields are always listed in hexadecimal notation.

The simple routine below, converting from ASCII to Binary, illustrates the list format.

```
##########################################################################
#                                                                        #
#    Archimedes  8051 Assembler V1.80/MD2      01/Jan/86  09:06:30      #
#                                                                        #
#       Source   =   exmpl.s03                                           #
#       List     =   prn.lst                                             #
#       Object   =   exmpl.r03                                           #
#       Options  =   s                                                   #
#                                                                        #
#                        (c) Copyright Archimedes Software, Inc. 1985    #
##########################################################################

   1  0000                        name    num_conversion
   2
   3  0000                        rseg    CODE
   4  0000                        public  ascbin
   5
   6  0030              zero      equ     '0'
   7  002B              plus      equ     '+'
```

```
 8  002D             minus    equ     '-'
 9  0000 A830        ascbin:  mov     r0,zero
10                                    ;Compute first digit, store R3.
11  0002 08                   inc     r0
12  0003 E6                   mov     a,@r0
13  0004 C3                   clr     c
14  0005 9430                 subb    a,#zero
15  0007 75F064               mov     b,#100
16  000A A4                   mul     ab
17  000B FB                   mov     r3,a
18  000C 08                   inc     r0
19  000D E6                   mov     a,@r0
20  000E 9430                 subb    a,#zero
21  0010 75F00A               mov     b,#10
22  0013 A4                   mul     ab
23  0014 2B                   add     a,r3
24  0015 FB                   mov     r3,a
25  0016 08                   inc     r0
26  0017 E6                   mov     a,@r0
27  0018 C3                   clr     c
28  0019 9430                 subb    a,#zero
29  001B 2B                   add     a,r3
30  001C FB                   mov     r3,a
31  001D E7                   mov     a,@r1
32  001E B42D04               cjne    a,#minus,pos
33  0021 EB                   mov     a,r3
34  0022 F4                   cpl     a
35  0023 04                   inc     a
36  0024 FB                   mov     r3,a
37  0025 EB          pos:     mov     a,r3
38  0026 F7                   mov     @r1,a
39  0027 22                   ret
40  0028             end

 Errors:  None      ##################
 Bytes:   40        # num_conversion #
 CRC:     DE2C      ##################

C>_
```

The header with assembly parameters as shown above, is only output on lists directed to other files than the terminal.

The item "Bytes" shows how much instruction code and constant data that was generated by the current module.

"CRC" contains a CRC checksum of the current module. Note that the CRC does not only depend on the source code, but also on *when* the assembly was performed, and if the 'S' option (or LOCSYM+) was used.

Because of the Archimedes assembler's multi-module capability (see section 2.14), the name of the current module is written (surrounded by #) after its last statement. This makes it easier to find a particular module in a long list of modules.

Macro generated lines will, if listed, have their source line number fields replaced with a slash ('/'). That is, numeric line number fields always show the actual number of source lines read from the input files.

2.8 Source Lines, Include Files and Label Syntax

A source line can consist of:

1. A blank line (a single <CR> or spaces or tabs followed by a <CR>).

2. A single comment. A semi-colon must precede the comment. (Spaces or tabs may precede the comment).

3. A single label in some cases followed by a comment. The label may be preceded by tabs or spaces. The label must be terminated by a colon. If a comment is used it must be preceded by a semicolon. The label will be assigned the value and type of the current location counter (PLC).

4. An assembler instruction or directive. See section 2.13 for details.

5. $<filename> This is a directive to the assembler which enables the user to *include* files in the source code. The dollar sign must be located at the first position of a source line, and no extra characters are allowed between the $ and the filename. The include file names follow the same rules as source files specified in the assembler start command and should be terminated with a <CR>. Include files can be nested to a maximum level of three. If nested include files are used the maximum number of open files should be set to 14 in **config.sys**. When an include file is found, the assembler switches its input to the specified file. After EOF has been detected in the included file, input resumes at the line after the $ directive of the previous file. Inclusion of files can be controlled with conditional assembly.

6. For an ordinary statement:

   a. The label or instruction may be preceded by tabs or spaces.
   b. If a label is used it must be separated from an instruction by colon, spaces or tabs.
   c. Instructions and operands may be separated by tabs or spaces.
   d. Operands must be separated by commas only.
   e. Comments must be separated from instructions or operands by a semicolon.

A source line must not exceed 255 characters.

Tabulators (ASCII 09H), are expanded according to the most common practice, that is, to columns 9, 17, 25 etc...

Label syntax. The first character in a label must be alphabetic (A-Z, a-z), or the special character "?" or "_". The following characters (if any) can be any of the already mentioned characters or the decimal digits (0-9). The <%> used in macro names is "%".

2.9 Symbols

User defined symbols can be up to 255 characters long, and all characters are significant. All instructions, registers and operators are reserved words and cannot be used as labels. The location counter symbol is recognized by a dollar character (e.g. JMP $+8). Predefined symbols (same as specified by Intel) are regarded as absolute values in expressions i.e. no typing (CSEG BSEG etc...) exists in this implementation. In user defined symbols upper and lowercase are significant, whereas this is not the case for predefined symbols like instructions, registers, operators etc.

```
My_own_symbol   &   MY_own_symbol       Are not equivalent

JMP   & ' jmp   &   jMp                 Are all valid forms of a
                                        JMP instruction
```

2.10 Integer, ASCII and Real Constants

Since the Archimedes assembler uses a 32-bit two's complement internal arithmetic, integers have a (signed) range from -2147483648 to 2147483647. This also means that ASCII constants (used in expressions) can be from zero to four characters long. ASCII constants are written as character strings enclosed with single quotes ('). Only printable characters and spaces may be used in ASCII strings. If the quote character itself is to be accessed, two consecutive quotes must be used:

```
'ABCD'          =>      ABCD
'A''B'          =>      A'B
'A'''           =>      A'
''''            =>      '
''                      =>      Empty string (value=0)
```

The assembler incorporates special syntax for floating point constants. The single precision 32-bit IEEE format is supported, and real constants will be regarded by the assembler as any other constant. Range is about 1E-38 to 1E38. Formal syntax:

```
" [<+|->] [<digit(s)>] . [<digit(s)>] [<E|e> [<+|->] <digit(s)>]
```

Some valid real constants:

```
"1.0    "100.    ".007    "-0.1e20          "25.0E12
"0.00000000000000000005          "1.0e-20          "0.0
```

Note that although the expression "1.5*"2.0 is not flagged as erroneous, it will not generate the correct representation of 3.0, because all operators treat values as integers. No spaces or tabs are allowed in real constants.

The Archimedes assembler recognizes the following types of constants:

'A'	=	ASCII constant
'AB"C'	=	ASCII constant
78	=	Decimal constant
56H	=	Hexadecimal constant
10110B	=	Binary constant
457Q	=	Octal constant
"1.0E3	=	Floating point constant

2.11 Expressions and Operators

Expressions consist of operands and operators. The valid operands in an expression are:

1. User defined symbols (labels).
2. Constants (numeric or ASCII).
3. The location counter (PLC) symbol.

The assembler accepts a wide range of expressions, including arithmetic and logical operations. All operations use a 32-bit, two's complement integer format, and range checking is only performed when a value is used to generate code. Expressions are evaluated from left to right, but the order also depends on the operator used.

Operator	Function	Order	Notes	Unary/Bin
-	Unary minus	1		Unary
NOT	Logical NOT	1		Unary
LOW	Low byte of	1	[1]	Unary
HIGH	High byte of	1	[1]	Unary
LWRD	Low word of	1	[1]	Unary
HWRD	High word of	1	[1]	Unary
DATE	Current time/date	1		Unary
SFB	Segment begin	1	[2]	Unary
SFE	Segment end	1	[2]	Unary
*	Multiplication	3		Binary
/	Division	3		Binary
+	Addition	4	[1]	Binary
-	Subtraction	4	[1]	Binary
MOD	Modulo	3		Binary
SHR	Logical shift right	3		Binary
SHL	Logical shift left	3		Binary
AND or &	Logical AND	5		Binary
OR	Logical OR	6		Binary
XOR	Logical XOR	6		Binary
EQ or =	Equal	7		Binary
NE or <>	Not equal	7		Binary
GE or >=	Greater or equal	7		Binary
LE or <=	Less or equal	7		Binary
GT or >	Greater	7		Binary
LT or <	Less	7		Binary
UGT or >>	Unsigned greater	7		Binary
ULT or <<	Unsigned less	7		Binary
SCON	Serial Connection	7		Binary

Spaces or tabs are ignored in expressions. The operators can be divided into arithmetic, logical, shift and special operators. Unary operators need an argument on the right side, while binary need arguments on both sides. Parantheses can be used to override the order of operators (e.g. LOW 1000+1000 is not equivalent to LOW(1000+1000)!). The operators noted with [1] can also be used (although with some limitations) on EXTERNal and relocatable expressions.

Special cases are segment functions [2], which return the start or end address of a loaded segment (plus or minus an offset). Segment functions can be used in code generating (i.e. not in EQU's), 8, 16 and 32-bit direct operands, as well as in 16-bit, PC-relative operands. Syntax:

```
SFB  (<segment> [<+|-> <offset>])
SFE  (<segment> [<+|-> <offset>])
```

Parantheses are optional if no offset is present (SFE CODE = SFE(CODE)).

The modulo (MOD) operator is defined as:

```
X MOD Y   =   X-Y*(X/Y)          using integer division
```

The shift operators (SHL and SHR) shift the first argument right or left by the number of positions given in the second argument. Zeroes are shifted into the high or low order bits, respectively. An example:

```
45 SHR 2          will generate a result equal to 11
```

The 8 comparison operators (EQ, NE, GE, LE, GT, LT, UGT and ULT) will evaluate to a logical true (all ones), if the comparison is true, else the result will be a logical false (all zeroes).

The operators LT, GT, GE and LE compare signed arguments, whereas UGT and ULT assume unsigned arguments.

The LOW, HIGH, LWRD and HWRD operators will generate the following result, if used on an absolute expression with the value 12345678 (in hexadecimal notation):

```
LOW     12345678=     78      (lowest 8 bits)
HIGH    12345678=     56      (second byte)
LWRD    12345678=     5678    (lowest 16 bits)
HWRD    12345678=     1234    (highest 16 bits)
```

Note that in the Archimedes assembler definition, the LOW, HIGH, LWRD and HWRD operators can also be used on EXTERNal and relocatable expressions, but only in *code* generating expressions of compatible length. Some examples will explain:

```
        EXTERN  REAL_NUMBER              ; An IEEE real constant maybe
        MOV     DPTR,#HWRD REAL_NUMBER   ; The highest 16 bits
        PUSH    DPL
        PUSH    DPH
        MOV     DPTR,#LWRD REAL_NUMBER   ; The lowest 16 bits
        PUSH    DPL
        PUSH    DPH                      ; Now 32 bits on the stack!

LABEL   EQU     LWRD REAL_NUMBER         ; NOT ALLOWED (not code-gen)
        MOV     R0,#LWRD REAL_NUMBER     ; NOT ALLOWED (8 versus 16 bits)
        MOV     R0,#LOW(REAL_NUMBER-4)   ; Ok
```

Note that the usage of parentheses, as in the last example, is mandatory on operands containing more than one item.

The DATE operator returns the moment when the current assembly begun. These values can also be obtained from the assembly list header, and this information could be used to create version data automatically.

The DATE operator takes an absolute argument (expression) and returns:

```
DATE  1  current second (0-59)
DATE  2  current minute (0-59)
DATE  3  current hour (0-23)
DATE  4  current day (1-31)
DATE  5  current month (1-12)
DATE  6  current year MOD 100   (1983 => 83)
```

2.12 Arithmetic Operations and Relocation

Besides a value, an expression also has a type associated to it. The possible types are:

Absolute
Relocatable (belongs to a RSEG, COMMON or STACK segment)
External

Not all operations are possible to use with relocatable and EXTERNal arguments. The allowed EXTERNal expressions are:

<external symbol> + <absolute expression>
<external symbol> - <absolute expression>

The result of one of these operations on an external symbol is a value (offset) with the type external.

The allowed *relocatable* expressions, and the resulting types are:

Operation	**X(rel),Y(rel)**	**X(rel),Y(abs)**	**X(abs),Y(rel)**
X + Y	invalid	relocatable	relocatable
X - Y	absolute	relocatable	invalid

When two relocatable arguments are involved they must belong to the *same* segment.

2.13 Directives

Assembler directives are special instructions which control various parts of the assembly process. The actions that can be controlled by the Archimedes assembler include list formatting, conditional assembly, separate assembly, memory allocation, macro definitions, and value assigments to symbols. The assembler support absolute, as well as relocatable code.
Typical uses of absolute code:

1. For *small* systems written as single modules.
2. To supply trap or interrupt vectors.

Relocatable code can be generated on any number of user defined segments. How many segments that should be used depends on the application. Simple assembly-based embedded microcomputer systems, where code is put in PROM's, the following definitions would probably be most useful:

```
RSEG  PROM      for program code and constant data
RSEG  RAM       for variables and stacks
```

Summary of the Archimedes assembler directives.

```
[<label>]     NAME     <symbol> [(<expression>)]          [<comment>]
[<label>]     MODULE   <symbol> [(<expression>)]          [<comment>]
[<label>]     END      [<expression>]                     [<comment>]
[<label>]     ENDMOD   [<expression>]                     [<comment>]
[<label>]     PUBLIC   <symbol> [<type>] [,...]           [<comment>]
[<label>]     EXTERN   <symbol> [<type>] [,...]           [<comment>]
[<label>]     LOCSYM   <+|->                              [<comment>]

[<label>]     ASEG                                        [<comment>]
[<label>]     RSEG     <segment> [(<alignment>)]          [<comment>]
[<label>]     COMMON   <segment> [(<alignment>)]          [<comment>]
[<label>]     STACK    <segment> [(<alignment>)]          [<comment>]
[<label>]     ORG      <expression>                       [<comment>]

<label>       SET      <expression>                       [<comment>]
<label>       EQU      <expression>                       [<comment>]
<label>       =        <expression>                       [<comment>]
<label>       DEFINE   <expression>                       [<comment>]

[<label>]     IF       <condition>                        [<comment>]
[<label>]     ELSE                                        [<comment>]
[<label>]     ENDIF                                       [<comment>]

[<label>]     MACRO    <%><name>                          [<comment>]
[<label>]     ENDMAC                                      [<comment>]
[<label>]     <%><name>        [<macro parameters>]       [<comment>]

[<label>]     LSTOUT   <+|->                              [<comment>]
[<label>]     LSTCND   <+|->                              [<comment>]
[<label>]     LSTCOD   <+|->                              [<comment>]
```

```
[<label>]       LSTEXP  <+|->                    [<comment>]
[<label>]       LSTMAC  <+|->                    [<comment>]
[<label>]       LSTWID  <+|->                    [<comment>]
[<label>]       LSTFOR  <+|->                    [<comment>]
[<label>]       LSTPAG  <+|->                    [<comment>]
[<label>]       PAGSIZ  <expression>             [<comment>]
[<label>]       PAGE                             [<comment>]
[<label>]       TITL    <'header'>               [<comment>]
[<label>]       PTITL   <'header'>               [<comment>]
[<label>]       STITL   <'subheader'>            [<comment>]
[<label>]       PSTITL  <'subheader'>            [<comment>]
[<label>]       LSTXRF                           [<comment>]

[<label>]       DB      <operands>               [<comment>]
[<label>]       DW      <operands>               [<comment>]
[<label>]       DD      <operands>               [<comment>]
[<label>]       DS      <operands>               [<comment>]
[<label>]       EXTRN   <operands>               [<comment>]
```

The optional labels will, (except for a few documented exceptions), assume the value and type of the current location counter (PLC). Operands denoted with <+|-> are of the counting type. Counting directives work as follows: Two uses of '-' on a particular directive must be succeeded by two uses of '+' in order to restore status to the original value. With counting directives you can have global control, while still using directives selectively locally.

The DB directive initializes code memory with byte values. The DW directive initializes code memory with a list of word (16 bit) values. The DS directive reserves space in byte units. DB, DW and DS work according to Intel specifications while DD (define double word) is an extension to Intel's assembler. DB, DW and DD operands (separated by commas) may be any valid absolute, relocatable or external expression. DB accepts ASCII strings as well. Ranges are -128 to 255, -32,768 to 65,535 and without limits respectively. EXTRN (Intel's specification) is equivalent to EXTERN in the Archimedes asssembler.

Sections 2.14-2.19 contain detailed descriptions of the other directives above.

2.14 Modules

Assembly of any number of modules from a single source file is a very useful Archimedes assembler feature. The main application is creation of libraries containing lots of small modules (like run time systems for high level languages), where each module also often represents a single routine (entry). With the multi-module facility you can significantly reduce the number of source and object files needed, without resorting to unnecessary "monolithic" libraries (e.g. refer to two routines and get twenty...).

The following rules apply to multi-module assemblies:

1. At the beginning of a new module all user symbols, except for DEFINEd and MACROs, are deleted, location counters cleared, and mode is set to absolute.

2 List control directives remain in effect throughout the assembly.

Below is an outline showing how multi-modules should be designed.

```
NAME/MODULE
 .
 .          Module #1
 .
ENDMOD
NAME/MODULE
 .
 .          Module #2
 .
ENDMOD
NAME/MODULE
 .
 .          Last module
 .
END
```

Note that END always must be used in the *last* module, and that there must not be any source lines (except for comments and list control directives), between an ENDMOD and a NAME/MODULE directive. In most cases the MODULE directive is preferable to the NAME directive, because the former generates the module attribute "library". However, object modules can, if needed, later have their attributes changed with the Archimedes librarian, XLIB.

```
[<label>]   NAME      <symbol> [(<expression>)]   [<comment>]
[<label>]   MODULE    <symbol> [(<expression>)]   [<comment>]
```

The NAME or MODULE directives are used to assign a name to the module and must be the first non-comment in the source code. NAME means that the module is considered as "program", whereas MODULE sets the module load attribute to "library". This has implications when the module is processed with XLINK (see chapter 3). If the NAME or MODULE directive is missing, the module will be assigned the name of the source file and the attribute "program". The symbol specified, in the NAME or MODULE directive, has no other function after the directive is used. That is, the symbol is left undefined, and may be used as any other symbol, including the declaration of the symbol as PUBLIC. The latter means that an entry and a module may have the *same* name.

After the name symbol, an absolute expression (with range 0-255), enclosed in parantheses (e.g. MODULE DOIO(10)) can optionally be specified. The expression value will be put in the LAN field (see section 3.13) of the module header, and will cause XLINK to do a default library search (while there are still unresolved externals) on a file named DFLTLB## (with the usual 8051 (.s03) dependent file extension), where ## is the hexadecimal representation of the expression value (00-FF). In the absence of an expression, the LAN field is set to zero.

```
[<label>]   END       [<expression>]   [<comment>]
[<label>]   ENDMOD    [<expression>]   [<comment>]
```

These directives terminate the assembly of the current module. END terminates the last module (often the only module).

If the optional expression is present, it will be output in the object code as a program entry address. Program entries must be either relocatable or absolute, and will show up in XLINK load maps, as well as in some of the hexadecimal absolute output formats.

```
[<label>]   PUBLIC      <symbol> [<type>] [,...]    [<comment>]
```

Makes one or more symbols available to other modules. The symbols declared as PUBLIC can only be assigned values by using them as labels. PUBLIC declared symbols can be relocatable or absolute, and can also be used in expressions (with the same rules as for other symbols). Note that the PUBLIC directive always "exports" full 32-bit values, which makes it feasable to use global 32-bit constants also in 8051 assembler. With the LOW, HIGH, LWRD and HWRD operators any part of such a constant can be loaded in a 8 or 16-bit register or word (see 2.11). The optional <type> record consists of 1-255 (absolute) expressions separated by commas and enclosed in parantheses. Example:

```
        PUBLIC  FOO(4,255,7)
```

The values (with range 0-255) fill the type fields of PUBLIC symbols and can be used to check validity of "interfaces" between modules. Type checking is performed by XLINK only on symbols which contain a type record. The PUBLIC directive can appear anywhere in the source code, but only one level of forward references is allowed:

```
PASS1_1:
        PUBLIC   PASS1_1,PASS1_2      Ok. (0 & 1 level forward ref.)
PASS1_2:
        PUBLIC   PASS2_1              Bad! (two level forward ref.)
PASS2_1 EQU      FORWARD
PASS2_2 EQU      FORWARD
        PUBLIC   PASS2_2              Ok. (one level forward ref.)
FORWARD:
```

There are no restrictions on the number of PUBLIC declared symbols in a module.

```
[<label>]   EXTERN      <symbol> [<type>] [,...]    [<comment>]
```

Defines symbols as external to the current module. The symbols defined as EXTERN can be used in expressions although the allowed operations are somewhat limited (refer to sections 2.11 and 2.12).

The EXTERN declaration must precede any usage of the symbols defined by it.

The optional <type> record is analogous with <type> in PUBLIC directives.

The maximum number of externally defined symbols within a single module is limited to 256.

```
[<label>]   LOCSYM      <+|->   [<comment>]
```

The LOCSYM+ directive, makes local symbols available in the object code, which will show up in XLINK load maps (equivalent to the 'S' option). Default is LOCSYM-.

2.15 Segments

```
[<label>]   ASEG                                        [<comment>]
[<label>]   RSEG        <segment> [(<alignment>)]       [<comment>]
[<label>]   STACK       <segment> [(<alignment>)]       [<comment>]
[<label>]   COMMON      <segment> [(<alignment>)]       [<comment>]
```

The ASEG, RSEG, STACK or COMMON directives set the current mode of the assembly. ASEG sets the absolute mode (which is by default active at the beginning of a module). RSEG, STACK and COMMON set the relocatable assembly mode, and the difference between these directives is described in section 3.9. The operand is any valid user defined symbol.

The optional alignment operand is an absolute expression (with range 0-31), enclosed in parantheses, and can be used to make segments align to any modulo two page boundary. That is, if the directive RSEG CODE(3) is used, the start address of segment "CODE" will be aligned (upwards) to the nearest 8 byte (2**3) page boundary. Note that only the *first* segment directive for a particular segment can contain an alignment operand. The default alignment on segments is one (2**0).

The assembler maintains separate location counters (initially set to zero) for all segments, which makes it possible to switch segments and mode at anytime without the need to save the current segment location counter.

The optional label will assume the value and type of the *new* location counter.

Up to 256 unique, relocatable segments may be defined in a single module.

```
[<label>]   ORG  <expression>     [<comment>]
```

Sets the location counter of the current segment according to the value of <expression>.

The result of the expression must be of the same type as the current segment, that is, it is not valid to use ORG 10 during RSEG, since the expression is absolute (ORG <location counter symbol>+10 is the correct way, to advance a relocatable location counter 10 positions). ORG expressions must not contain any forward references.

The optional label will assume the value and type of the new location counter.

All location counters (PLC:s) are by default set to zero at the beginning of an assembly module.

2.16 Equates

```
<label>   SET        <expression>   [<comment>]
<label>   EQU        <expression>   [<comment>]
<label>   =          <expression>   [<comment>]
<label>   DEFINE     <expression>   [<comment>]
```

Equate directives assign values to symbols specified in the label fields. In addition to the value some other qualities of the expression are assigned to the symbol:

1. Expression type (absolute, relocatable, or EXTERNal)

2. A flag which tells if the expression contains any not yet defined symbols.

3. If the directive is a SET the symbol is marked as variable, which means that it may be redefined.

EQU and '=' are equivalent, and are the normal way of assigning values to symbols. Absolute and relocatable symbols, defined with these directives, can also be declared as PUBLIC. This is frequently very useful for defining fixed global addresses, indexes etc.

A difference compared to Intel's assembler is that registers are not allowed in expressions (i.e. you cannot write : TEMP EQU A).

One level of forward references is allowed in equate directives:

```
LABEL   EQU   3          Ok, no forward refs.
FCBDT   EQU   LABEL      Ok, no forward refs.
ERROR   EQU   FOO        Bad! Two levels of forward refs.
FOO     EQU   BAR        Ok, one level of forward ref.
BAR:
```

The SET directive allows symbols to be redefined, a feature that has its greatest value for defining macro variables, like counters etc. Symbols defined with SET cannot be PUBLIC declared.

The DEFINE directive is a special form of EQU, because symbols that are DEFINEd will remain known between modules. This is not to be confused with the PUBLIC directive which only makes symbols available to other modules at *link* time, while the DEFINE directive makes symbols available to other modules at *assembly* time (although DEFINEd symbols may also be declared as PUBLIC). Only absolute expressions are allowed in the DEFINE directive. The main application for DEFINE is to make configuration data known to other modules in an assembly. This could for example be used with conditional assembly.

Because this implementation also tranfers the type of the expression, a label can be set (using EQU, = or SET) equal to an EXTERNal symbol with an optional displacement added to it. All further use of the assigned symbol will be treated as references to the external symbol in the expression. This will also show up in cross reference lists.

2.17 Conditional Assembly

```
[<label>]   IF          <condition>     [<comment>]
[<label>]   ELSE                       [<comment>]
[<label>]   ENDIF                      [<comment>]
```

The conditional assembly facilities make it possible to control the assembly process at assembly time, using the directives, IF, ELSE, and ENDIF. The effect of an IF directive is that if the condition is not true, the code that follows will not generate any code (i.e. it will not be assembled or syntax checked) until an ENDIF or ELSE directive is found.

Conditional assembler directives may be used anywhere in an assembly, but have their greatest use in conjunction with macro processing.

The <condition> can be either:

1. An absolute expression

   The expression must not contain forward references, and any non-zero value is considered as true.

2. 'string1'='string2'

If string1 and string2 have the same length and contents, the condition is said to be true. Frequently usable in macros.

3. 'string1'<>'string2'

The opposite of 2.

All assembler directives (except for END) as well as file inclusion may be disabled by the conditional directives.

An IF directive must be terminated by an ENDIF directive, whereas the ELSE directive is optional, and if used, it must be inside of an IF-ENDIF block. IF-ENDIF and IF-ELSE-ENDIF blocks may be nested to any level.

Example 1. Simple IF-ENDIF Block:

```
IF  OPTION=3
.
.       Assembled if OPTION=3
.
ENDIF
```

Example 2. IF-ELSE-ENDIF Block:

```
IF DSKFLAG
.
.       Assembled if DSKFLAG <> 0
.
ELSE
.
.       Assembled if DSKFLAG = 0
.
ENDIF
```

If a label is used, it will be assigned the current value and type of the location counter, only if the assembly is not disabled *after* the directive is in effect, otherwise the label will remain undefined.

2.18 Macro Processing

With macros it is possible to generate often needed sequences with a single directive, though different parameters may be required. The macro facility is illustrated by the example below. The macro definition is on lines one thru six. The macro is invoked on line eleven. **Review section '2.21 Macro Examples' for two more elaborated examples using a recursive help macro to achieve a variable macro.**

```
#########################################################################
#                                                                       #
#    Archimedes  8051 Assembler V1.80/MD2      01/Jan/86  09:06:30      #
#                                                                       #
#       Source   =   makex.s03                                          #
#       List     =   prn.lst                                            #
#       Object   =   makex.r03                                          #
#       Options  =   x                                                  #
#                                                                       #
#                        (c) Copyright Archimedes Software, Inc. 1985   #
#########################################################################

   1  0000                    macro %moves
   2                          mov a,@r1
   3                          mov @r0,a
   4                          inc r1
   5                          inc r0
   6  0000                    endmac
   7  0000 D0E0               pop acc
   8  0002 F9                 mov r1,a
   9  0003 D0E0               pop acc
  10  0005 F8                 mov r0,a
  11  0006                    %moves
   /  0006 E7                 mov a,@r1
   /  0007 F6                 mov @r0,a
   /  0008 09                 inc r1
   /  0009 08                 inc r0
  12  000A                    end

  Errors:  None              #########
  Bytes:   10                # makex #
  CRC:     3345              #########
```

The macro facility incorporated in the Archimedes assembler can be characterized as a built-in pre-processor, in the sense that the macro expander does not know very much about the 8051 target processor, rather it has its own low-level "language".

On the following pages there is a formal description of the macro directives and operators. After that a few examples on how to use some of the more exotic functions.

```
[<label>]   MACRO        <%><name>    [<comment>]
```

The MACRO directive defines the name of the macro. It is also the first directive of a macro definition.

The <%> is used to separate macro names from other assembler symbols. Macro names may contain any valid symbol characters (see label syntax section 2-8 for details).

A macro may not be redefined.

Nested macro *definitions* are not allowed, but macros may call any other macros including calls to themselves (i.e. recursive macros). The maximum nesting level on macro *calls* is implementation dependent, but (if no parameters are used) always greater than 100.

Note that macro definitions are not cleared between modules in a multi-module assembly, although all variables (labels outside of the macro definitions) are lost.

```
[<label>]   ENDMAC    [<comment>]
```

The ENDMAC directive is the last statement of a macro defintion.

```
[<label>]   <%><name>   [<macro parms>]    [<comment>]
```

This is the macro call, and it can have up to 10 parameters, which can be referred to within the macro body (with \0 to \9). The macro parameters may contain any characters, but comma "," is assumed to be the delimiter between parameters. The standard comment delimiter or <CR> denotes the end of the argument list.

A null parameter may be written as ',,'. References (with \n operators) to non-existent parameters will yield a null string (.i.e not an error condition). When a macro parameter contains commas, a special syntax must be used and the macro parameter must be surrounded by angle brackets.

A call like this:

```
%D010    P0,<S,50>,,6
```

Would result in these parameter values:

```
\0     =      P0
\1     =      S,50
\2     =      null string
\3     =      6 ,
\4-\9  =      null string
```

In addition to the already mentioned \n operators which contain the actual text used by the current call, a macro expression stack (MSTACK) is also available. With the stack and a large set of operators, very complex macros can be defined. The macro stack is a global static array of integers (with the bounds 0 and 100). That the array is global and static means that it can be used for communication between different macros. Coupled with the macro stack is a stackpointer (SP), which is set to zero at the beginning of an assembler module. An additional part of the macro processor is a global integer (GENLAB) used for generating unique labels (cleared at the beginning of an assembler module). With these variables and the standard assembler directives, it is relatively easy to implement structured language elements (e.g. REPEAT-UNTIL, WHILE-ENDWHILE etc.).

Before digging too far into the macro jungle, it is important to know what the macro processor does and what the assembler does (i.e. though they are integrated into a single program, they are not tightly coupled).

There are three distinct phases in the macro process:

1. Scanning and saving of macro definitions is performed by the assembler. The text between MACRO and ENDMAC is saved but not syntax checked. Include files ($<file>) work as usual during macro scanning, but lines with the include directive are not saved (i.e. these lines will have no effect during macro *expansion*).

2. A macro call forces the assembler to invoke the macro processor (expander) which switches (if not already in a macro) the assembler input stream from a source file to the output from the macro expander (which takes its input from the requested macro definition). The macro expander has no knowledge of assembler symbols since it only deals with text substitutions at source level. However, the current conditional assembly flag can be read (and acted upon) with \T and \F operators. Before a line from the called macro defintion is handed over to the assembler, the expander scans the line for all occurrences of \-operators, and replaces them with their return values. That is, the *assembler* does not understand \-operators, rather it will flag them as erroneous.

3. The expanded line is then processed as any other assembler source line. The input stream to the assembler will continue to be the output from the macro processor, until all lines of the current macro definition have been read.

Summary of MACRO substition operators

Operator	Description	Return Value
\0 - \9	Current macro call parameters	The actual text (t)
\?	Push number of parameters in call	New TOS (n)
\\	Insert a '\'	\ (t)
\Lnnn	Push constant "nnn"	A null string (t)
\Lnnn.	Syntactic variation of the former	A null string (t)
\@	Push GENLAB, increment GENLAB	New TOS (n)
\.	Get TOS without popping	TOS (n)
\<	Get current TOS and pop stack	Previous TOS (n)
\O	Pop stack	New TOS (n)
\A	Swap TOS and NOS	New TOS (n)
\U	Push TOS (copy to NOS)	TOS (n)
\=	SP:=TOS	New SP (n)
\$	Push value of SP	New TOS (n)
\%	TOS:=length_of(\n)	New TOS (n)
\"	Get substring	Substring (t)
\S	Store value indirect	Value (n)
\R	Restore (get) value indirect	Value (n)
\N	Negate TOS	New TOS (n)
\I	Increment TOS	New TOS (n)
\D	Decrement TOS	New TOS (n)
\+	TOS:=TOS+NOS	New TOS (n)
\-	TOS:=TOS-NOS	New TOS (n)
*	TOS:=TOS*NOS	New TOS (n)
\/	TOS:=TOS/NOS	New TOS (n)
\T	Expand only if cond-flag is true	A null string (t)
\F	Expand only if cond-flag is false	A null string (t)

This table is a summary, and on the following pages there are more detailed explanations of these operators. The "(n)" after a return value description, means a numeric (decimal) value, while "(t)" denotes a text string. Note that alphabetic values can either be written in upper or lowercase (i.e. \o is equal to \O).

TOS is equal to Top Of Stack (MSTACK[SP]). NOS is equal to Next On Stack (MSTACK[SP-1]).

\0 - \9

Returns: The actual text in the parameters of the current macro call.

\?

SP:=SP+1
MSTACK[SP]:=Number_of_parameters_in_current_macro_call.

Returns: Numeric_string_of(TOS)

****

Since the backslash is interpreted as a command (only during macro expansion), this operator makes it possible to insert this charcter in the text without it being treated as a command.

Returns: \

\Lnnn
\Lnnn.

SP:=SP+1

MSTACK[SP]:=nnn

Returns: Nothing (the null string) .

The constant ("nnn") is a decimal number in the range of 0 to 9999. The variant succeeded by a period makes it possible to place an \L operator immediately before a numeric constant.

\@

SP:=SP+1

MSTACK[SP]:=GENLAB
GENLAB:=GENLAB+1

Returns: Numeric_string_of(TOS).

\.

Returns: Numeric_string_of(TOS)

\<

Tempval:=MSTACK[SP]

SP:=SP-1

Returns: Numeric_string_of(Tempval)

\O

SP:=SP-1

Returns: Numeric_string_of(TOS)

\A

MSTACK[SP] <--> MSTACK[SP-1]

Returns: Numeric_string_of(TOS)

\U

SP:=SP+1
MSTACK[SP]:=MSTACK[SP-1]

Returns: Numeric_string_of(TOS)

\=

SP:=TOS

Returns: Numeric_string_of(SP)

\=

SP:=TOS

Returns: Numeric_string_of(SP)

\$

MSTACK[SP+1]:=SP
SP:=SP+1

Returns: Numeric_string_of(TOS)

\%

MSTACK[SP]:=Length_of_parameter_no(MSTACK[SP])

Returns: Numeric_string_of(TOS)

\"

Parameter_no:=MSTACK[SP]
Start_of_string:=MSTACK[SP-1]
End_of_string:=MSTACK[SP-2]
SP:=SP-3

Returns: This operator returns the text of the requested parameter (0-9), from "Start_of_string" to "End_of_string". A string is assumed to begin with index 1. If Start_of_string > End_of_string the returned string will be reversed. If both string pointers are zero, the null string will be returned.

\S

```
Tempval:=MSTACK[SP]
SP:=SP-1
MSTACK[MSTACK[SP]+SP]:=Tempval
SP:=SP-1
```

Returns: Numeric_string_of(Tempval)

\R

```
MSTACK[SP]:=MSTACK[MSTACK[SP]+SP]
```

Returns: Numeric_string_of(TOS)

\N
\I
\D

```
N:      MSTACK[SP]:=-MSTACK[SP]
I:      MSTACK[SP]:=MSTACK[SP]+1
D:      MSTACK[SP]:=MSTACK[SP]-1
```

Returns: Numeric_string_of(TOS)

\+
\-
*****
\/

```
MSTACK[SP-1]:=TOS op NOS
SP:=SP-1
```

Returns: Numeric_string_of(TOS)

\T
\F

At the beginning of a line, the macro expander will process all \-operators. This can be partially disabled by the \F and \T operators. When one of these operators is found, further processing of the *other* \-operators is disabled, depending on the value of the current conditional assembly flag.

\T -> disable if false conditional flag.
\F -> disable if true conditional flag.

Returns: Nothing (the null string)

2.19 List Control

```
[<label>]   LSTOUT      <+|->   [<comment>]
```

LSTOUT+ enables assembly list output (if a list file was specified), and is by default active. LSTOUT- disables all list output except for error messages and also overrides all other list control directives. Both forms of LSTOUT are *invisible* in assembly lists.

```
[<label>]   LSTCND      <+|->   [<comment>]
```

LSTCND+ will force the assembler only to list source code for the parts of the assembly that are not disabled by previous conditional statements (IF, ELSE or ENDIF). LSTCND- is by default active and makes the assembler list all source lines.

```
[<label>]   LSTCOD      <+|->   [<comment>]
```

LSTCOD+ will expand the listing of output code to more than one line if needed (i.e. long ASCII strings will produce several lines of list output because of the great amount of code they generate) and is by default active. LSTCOD- will only list the first line of code for a source line, but code generation is *not* affected.

```
[<label>]   LSTEXP      <+|->   [<comment>]
```

LSTEXP+ makes all macro *generated* lines listed, and is by default active. With LSTEXP- listing of macro generated lines can be disabled. Both forms of LSTEXP are *invisible* in assembly lists.

```
[<label>]   LSTMAC      <+|->   [<comment>]
```

LSTMAC+ makes all macro *definitions* listed, and is by default active. With LSTMAC- listing of macro definitions can be disabled. Both forms of LSTMAC are *invisible* in assembly lists.

```
[<label>]   LSTWID      <+|->   [<comment>]
```

LSTWID- formats the assembly output list in the following way (default active):

```
<TL><AF><CF><SF>.
```

LSTWID+ formats the assembly output list in the following way:

```
<TL><NL><LL><AF><CF><SF>

<TL> ::=Total number of lines of source read
<NL> ::=Nesting level of files
<LL> ::=Local line number in current file
<AF> ::=Address field
<CF> ::=Data field
<SF> ::=Source field
```

```
[<label>]   LSTFOR      <+|->   [<comment>]
```

LSTFOR- makes the source lines list "as is", although tabulators are expanded, and is by default active. LSTFOR+ will make a "nice" assembly list with formatted fields, regardless of the shape of the input file.

```
[<label>]   LSTPAG      <+|->   [<comment>]
```

LSTPAG+ formats the assembly output list into pages.
LSTPAG- suppresses paging of the assembly list, and is by default active.

```
[<label>]   PAGSIZ      <expression>    [<comment>]
```

PAGSIZ sets the number of printed lines per page of the assembly list, according to the expression (which must be absolute, and in the range of 10-150). The number of lines per page is by default set to 44.

```
[<label>]   PAGE                    [<comment>]
```

PAGE will generate a new page in the assembly list if paging is active. The PAGE directive is *invisible* in assembly lists.

```
[<label>]   TITL <'header'>       [<comment>]
```

TITL fills the page header with the string in the operand field. The header string will be printed on the top of every page. At the beginning of an assembly the header string is blank.

```
[<label>]   STITL       <'subheader'>   [<comment>]
```

STITL fills the page subheader with the string in the operand field. The subheader string will be printed on the line below the header of every page. At the beginning of a new module the subheader string is blank.

```
[<label>]   PTITL       <'header'>      [<comment>]
```

PTITL is equivalent to TITL+PAGE, and is not listed.

```
[<label>]   PSTITL      <'subheader'>   [<comment>]
```

PSTITL is equivalent to STITL+PAGE, and is not listed.

```
[<label>]   LSTXRF                      [<comment>]
```

LSTXRF will generate a cross reference table at the end of the assembly list for the current module. The table displays besides values and line numbers, also the type of the symbol. Types:

Xhh	EXTERNal symbol definition "hh".
A	Local absolute symbol.
A*	Local DEFINEd absolute symbol.
Rhh	Local relocatable (to segment "hh") symbol.
AE	PUBLIC absolute symbol.
AE*	PUBLIC absolute and DEFINEd symbol.
RhhE	PUBLIC relocatable (to segment "hh") symbol.
ShhR	RSEG segment defintion "hh".
ShhS	STACK segment defintion "hh".
ShhC	COMMON segment defintion "hh".
V	Variable type (SET has been used).
U	Undefined symbol.
	Macro definitions are untyped.

"hh" = relative declaration number (external/segment definitions).

2.20 Sample Session

The example below illustrates the use of the assembler and the linker. It starts with an assembly using the symbol and cross-reference table option on. The linker is then invoked and finally the produced object code is listed. The example is the same as used earlier in the chapter (p. 2-6). It is somewhat superficial (e.g. linking only one module), however, the objective is to show the use of the directives, commands and options.

```
A8051 exmpl prn s

#######################################################################
#                                                                     #
#    Archimedes  8051 Assembler V1.80/MD2      01/Jan/86  09:06:30    #
#                                                                     #
#       Source   =   exmpl.s03                                        #
#       List     =   prn.lst                                          #
#       Object   =   exmpl.r03                                        #
#       Options  =   s                                                #
#                                                                     #
#                      (c) Copyright Archimedes Software, Inc. 1985   #
#######################################################################

   1  0000                        name    num_conversion
   2
   3  0000                        rseg    CODE
   4  0000                        public  ascbin
   5
   6  0030              zero      equ     '0'
   7  002B              plus      equ     '+'
   8  002D              minus     equ     '-'
   9  0000 A830         ascbin:   mov     r0,zero
  10                               ;Compute first digit, store R3.
  11  0002 08                     inc     r0
  12  0003 E6                     mov     a,@r0
  13  0004 C3                     clr     c
  14  0005 9430                   subb    a,#zero
```

```
15  0007 75F064             mov     b,#100
16  000A A4                 mul     ab
17  000B FB                 mov     r3,a
18  000C 08                 inc     r0
19  000D E6                 mov     a,@r0
20  000E 9430               subb    a,#zero
21  0010 75F00A             mov     b,#10
22  0013 A4                 mul     ab
23  0014 28                 add     a,r3
24  0015 FB                 mov     r3,a
25  0016 08                 inc     r0
26  0017 E6                 mov     a,@r0
27  0018 C3                 clr     c
28  0019 9430               subb    a,#zero
29  001B 28                 add     a,r3
30  001C FB                 mov     r3,a
31  001D E7                 mov     a,@r1
32  001E B42D04             cjne    a,#minus,pos
33  0021 EB                 mov     a,r3
34  0022 F4                 cpl     a
35  0023 04                 inc     a
36  0024 FB                 mov     r3,a
37  0025 EB         pos:    mov     a,r3
38  0026 F7                 mov     @r1,a
39  0027 22                 ret
40  0028                    end

 Errors:  None      ################
 Bytes:   40        # num_conversion #
 CRC:     DE2C      ################
```

```
                    Symbol and Cross Reference Table
                    ================================
 Symbol     Value    Type  Defline   Refline
                          Segment Definitions
                          ===================
 CODE       0028     SOOR      3
                          External Symbols
                          ================
                          Public Symbols
                          ==============
 ascbin     0000     ROOE      9         4
                          Local Symbols
                          =============
 B          00F0     A        -1        15       21
 minus      002D     A         8        32
 plus       002B     A         7
 pos        0025     ROO      37        32
 zero       0030     A         6         9       14       20       28
                          Macro Definitions
                          =================
xlink 8051 exmpl /x

        Archimedes Linking Loader V2.00/MD2
    (c) Copyright Archimedes Software, Inc. 1985

    Segments          Loc       Siz       Typ    Org    P/N    Al
    ========          ===       ===       ===    ===    ===    ==
    CODE              0000      0028      Rel    Flt    Pos    01

    Modules/Entries          Values             References
    ===============          ======             ==========
    num_conversion    (UP)
                        E  0000/CODE
   Link complete, no warnings.

type aout.a03
:10000000A83008E6C3943075F064A4FB08E6943089
:1000100075F00AA42BFB08E6C394302BFBE7B42D44
:0800200004EBF404FBEBF722F2
:00000001FF
```

The resulting code (here in Intel hex format), can be downloaded to debuggers or emulators, or "burned" into PROM's.

2.21 Macro Examples

Below two more elaborated macro examples are listed. The first example shows how to create a repeat function through the use of a recursive macro.

```
 1   * MACRO EXAMPLE ON HOW TO CREATE A REPEAT FUNCTION
 2   * USE OF A RECURSIVE MACRO
 3
 4
 5   0000            macro   %repeat
 6                           if \0>0
 7                   rep_count
 8                           endif
 9                   rep_count       set     rep_count+1
10                           if rep_count<=0
11                           lstexp  +
12                           \1
13                           lstexp  -
14                           %repeat -1,\1
15                           endif
16   0000                    endmac
17
18   0000                    %repeat 3,NOP
 /0000 01                   NOP
 /  0001 01                 NOP
 /  0002 01                 NOP
19   0003                    end

 Errors:  None              ##########
 Bytes:   3                 # repeat #
 CRC:     4BA2              ##########
```

The macro example on the next page shows how to achieve a variable macro through the use of a recursive help macro. This scheme is often required as the macro expander does not have direct access to the symbol table.

```
1   * MACRO EXAMPLE ON HOW TO GET A VARIABLE MACRO THROUGH THE
2   * USE OF A RECURSIVE HELP MACRO.  THIS SCHEME IS OFTEN REQUIRED
3   * DUE TO THAT THE MACRO EXPANDER DOES NOT HAVE DIRECT ACCESS
4   * TO THE SYMBOL TABLE.
5
6   0000                    macro   %set_MSTACK
7                           lstexp  -
8                           if \0>=0        Init call?
9                   * \t\l1
10                  ?count  set     -(\0)
11                          endif
12                  ?count  set     ?count+1
13                          if      ?count<0
14                  * \t\i
15                          %set_MSTACK     -1
16                          endif
17                          lstexp  +
18  0000                    endmac
19
20  0000                    macro   %substr
21                          lstexp -
22                          %set_MSTACK     \1
23                          %set_MSTACK     \0
24                  * \l2
25                          lstexp  +
26                  *       \"
27  0000                    endmac
28  0004            start   set     4
29  0007            endstr  set     7
30  0000                    %substr 2,4,abcdefg
 /                  *       bcd
31  0000                    %substr start,endstr,abcdefg
 /  0000            *       defg
32  0000                    end


Errors:  None           ##########
Bytes:   0              # substr #
CRC:     0429           ##########
```

2.22 Intel ASM Compatibility

The Archimedes macro assembler follows the implementation of the Inte 8051 assembler closely. The most significant difference relative to the Intel assembler are in macro and segment definitions. In macros the "%"-sign is placed differently. The Intel assembler uses a two-step process in first defining segments and then using certain segment operators. The Archimedes assembler uses only ASEG, RSEG, STACK and COMMON segment definitions and has no support of bit segments and does not allow an AJMP or ACALL in an RSEG segment even when the destination address is absolute.

It's quite straightforward to preserve any Intel assembly code investment. First of all, existing Intel ASM51 code must be reassembled since the relocatable object module formats created by Intel ASM51 and the Archimedes assemblers and compilers are different. In reassembling your ASM51 source code with the Archimedes assembler, A8051, you will find that most source statements, labels, constants, basic operators, etc. are compatible with Intel ASM51. However, there are a few things that may need to be changed; some of these are briefly described below.

Segment Directives: The use of segment directives differs slightly in that Intel ASM51 requires that you explicitly define a segment name and its type before using it while Archimedes A8051 does not:

```
INTEL                                   ARCHIMEDES

CODE_SEG    SEGMENT CODE        ; no need to pre-define
DATA_SEG    SEGMENT DATA        ;   segment names
            .
             RSEG   CODE_SEG    RSEG        CODE_SEG
            . (code)                    . (code)
             RSEG   DATA_SEG    RSEG        DATA_SEG
            . (data)                    . (data)
```

Bit Segments: Archimedes A8051 does not support "bit" segments (the "BSEG" segment type) or the "BIT" or "DBIT" ASM51 directives. Bit addressable RAM locations (internal addresses 20-2Fh) can be reserved in A8051 as shown below:

```
        ASEG                    ; set an absolute segment
        ORG     20H             ;  anywhere in bit-addressable RAM
control:ds      1               ; reserve a byte
direc:  equ     control.0       ; define bits within byte...
alarm:  equ     control.1
        .
status: ds      1               ; reserve next byte...
mode:   equ     status.0
        .
        RSEG    CODE
        setb    direc
        jb      alarm,task1
        .
```

By the way, if you need to access these bit locations from C, their absolute bit addresses must be explicitly defined because the set_bit, read_bit, clear_bit and read_bit_and_clear in-line C functions need constant expressions for the bit address:

```
#define DIREC 0         /* 1st bit in byte at 20H */
#define ALARM 1         /* 2nd bit       "        */
#define MODE 8          /* 1st bit in byte at 21H */
.
set_bit(ALARM);
if (read_bit(MODE))
{
        .
```

AJMPs and ACALLs: The Archimedes assembler does not permit the use of the two-byte AJMP and ACALL instructions within a relocatable segment (RSEG directive) as the linker is not able to check the range of the destination address. Use LJMP, SJMP, or LCALL as appropriate. AJMP and ACALL can be used in absolute segments (ASEG directive).

Macros: The Archimedes A8051 assembler contains a powerful macro processing capability. However, as it is not compatible with the Intel ASM51 macro conventions, programs that make use of macros will need some rewriting. The most important difference is that the Archimedes macro preprocessor does not access the symbol table, so that it is not possible to pass a value by its symbolic name. For example, see the macro call shown below:

```
        VALUE   EQU             8
        %MY_MAC VALUE                   ; does NOT work with A8051
```

Note that the parameter being passed to MY_MAC would equal "VALUE", not "8", after the Archimedes macro preprocessor does the substitution.

3.1 Introduction

The Archimedes linker XLINK is a very flexible program. It is equally suitable for converting output from simple absolute assembler programs to hex code as for linking and generating a load module for a complex multi-segmented system written in a high level language.

An internal 32-bit architecture gives XLINK the power to deal with present as well as future devices. Default libraries for high level languages are automatically loaded, and a built-in command file facility as well as configuration setups by environment variables, makes XLINK easy to use. However, there are many different options available so you are recommended to review the documentation closely. **In addition to this chapter, you should also review the tutorial chapter as well as section 1.13 Linking in the C-chapter.**

Errors and warnings are always expressed as complete messages and will thus seldom need any references to the manual. Please review Appendix F for details.

3.2 Operating Instructions

To start XLINK:

XLINK [*-Options*] *Input_file(s)* [*-Options*]

The command line arguments may be in any order. Upper or lower case is of importance in two cases.

1. The letter defining the switch (after the '-').
2. In symbol and segment names

One of the most useful options is the extended command line option, -f, which enables the use of a command file with suitable "default" options set. (The supplied linker command file is described in section 1.13 Linking in the C-chapter).

If XLINK is installed on a VAX/VMS system, the usage of quotation marks (") is necessary when using uppercase letters.

3.3 An Example

First a trivial example, where XLINK is used to transform a single absolute 8051 assembler program into hex code:

XLINK **-c8051 myfile**

Here no options were specified, and the following default values were used:

Default file extension on input files: R03 (CPU dependent).
Default output file: AOUT (Always, if no other was specified).
Default file extension on output file: A03 (CPU or format dependent).
Default output format: INTEL-STANDARD (CPU dependent).

The next example is a little bit more advanced. Two object files are linked,which contain modules using two relocatable segments named CODE and DATA. CODE is supposed to be put in EPROM with start at location zero. DATA is assumed to mirror the variable area located at RAM address 4000 (hex). For documentation purposes a cross reference list with segment and module map (option **-xsm**), called *main.lst* (option **-l**), is produced.

XLINK -c8051 main iolib -ZCODE=0 -ZDATA=4000 -xsm -l main

For more specific information on the available options, review the listing of all the commands in the next section.

Please note that space or tab are the only valid delimiters between options, while comma is used to separate items within an argument.

3.4 Linker Switch Commands

The linking process can be controlled by an extensive set of command line switches. The standard options available in the Archimedes Universal Linker are listed in this section.

Upper or lower case is of importance in two cases. First, the letter defining the switch (after the '-'). Second, in symbol and segment names. (If XLINK is installed on a VAX/VMS system, the usage of quotation marks (") is necessary when using uppercase letters). The position in the command line of switch options is of no importance. Please note that switch commands which have file(s) argument(s) must have the filename(s) separated from the switch flag with tabs or spaces. All other type of arguments should start immediately after the switch character. Spaces or tabs are the only valid delimiters between options, while comma is used to seperate items within an argument.

When a number is used in an option (an address, a value or such), a *decimal* or a *hexadecimal* number may be specified. If the number is preceded by a dot it is regarded as a decimal number, otherwise it is interpreted as hexadecimal.

-o file Put output on: <file> <.yyy>.

By default the linker will generate the output format on '*aout.yyy*' where yyy is a CPU- or format-dependent extension. The **-o** switch forces the linker to generate the format on any file. If a format which generates two output files is chosen, then the extension *yyy* will affect the primary file (first format).

-l file Generate a list on: <file> <.lst>.

If a list file is wanted, use this option. Filenames are treated analogous with the **-o** switch. Also see **-x** option.

-G	Disable global typecheck. With the -G switch the linker enters a mode where it doesn't check the *types* of the externals versus the entries.
-w	Disable warnings. If the -w flag is given in the command line, warning messages will not be displayed, although they will still be counted and in the end show up in linking statistics.
-S	Silent operation of linker. Makes linker turn off sign-on message and linker statistics report, i.e nothing appears on the screen. It does not disable the list facility (option -l).
-R	Disables range check. If an address is relocated out of the cpu's address range, no error message is generated.
-n	Disregard all locals. This option speeds up the linking process, but is only recommendable if locals is of no use.
-d	Disable code generation. No code will be generated.

-z	Disable overlay check. Do not check if the segments overlap each other.
-m	Enable file bound processing. This mode of processing uses less main memory. If the linker runs out of memory, this option will help. For more information see *3.12 File Bound Processing*.
-K	Suppress the loading of default libraries **DFLTLB??.r??**. Also see section *3.9 Default Libraries*.
-B	Forced dump. The linker will generate code, regardless of any entries or duplicated declarations of program modules.
-!	Comment until next '-!'.
-x{smi}	Generate cross-reference list.

When the -x switch is given, without any parameters, a list with the following is produced:

A header with information about target cpu, name of list file (if any), output format and command line. Address and module name for the program entry. A *module map* with all the absolute entries and locals, and a list of the module's segments with respective entries and locals. A *segment map* with all the segments, in dump order, start and end address, type, origin, allocation (up or down), alignment. For further information see *3.14 Cross Reference*. If the -x switch is succeeded by one or more of the arguments **s**, **m** and **i** the list is specified. It will only generate output according to the specification:

s: The segment map

m: The module map

i: The index, a list of all 'names' in the cross reference and the page they appear on.

Note if -l option is not specified the output will occur on the terminal

---

-pnn Page the listing with 'nn' lines/page (10-150).

nn is always a decimal number. Default is 80.

-ccpu Choose one of the following cpu's:

z8	8051	6801	6502
z80	8085	6805	t99
z8002	8086	6809	t7000
1802	8096	68k	7800
6301	16032	68hc11	

-Fformat Choose one of the following formats:

aomf8051	hp	mpds-i	pentica-a
aomf8096	hp-code	mpds-m	pentica-i
ashling	hp-symbol	mpds-sym	pentica-m
ashling-6301	intel-ext	msd	rca
ashling-64180	intel-std	msd-i	symbolic
ashling-6801	millenium	msd-m	tektronix
ashling-8080	motorola	msd-t	texas-instr
ashling-8085	mpds	nec	typed
ashling-z80	mpds-code	nec-symbolic	

-A file(s) Forced load of **file(s)**.

The modules in **file(s)** are considered to be program modules. They will always be loaded.

-C file(s) Conditional load of **file(s)**.

The modules in **file(s)** are considered to be library modules. They will only be linked if they are necessary, i.e. referred to by a module that is to be loaded.

-E file(s) Empty load of **file(s)**.

The modules in **file(s)** are linked as usual modules, but they are not dumped out in the resulting code.

-f file Extend command line with <file> <.xcl>

The command line to the linker can be extended to almost any length by using a command file which should contain command line parameters formatted in the same fashion as the ordinary command line. Also, see section *3.8 Command Files*.

-b(type)segs=strt,len,inc

Bank the segments in **segs**.

This option enables banking. **(type)** is optional (see *3.5 Define Segment*). **'segs'** is a list of segments, with comma or colon as delimiters. **'strt'** is the start address for the banks, i.e where the first bank starts. **'len'** is the length of a bank **'inc'** is the incremental factor , i.e on address **start+inc** the second bank begins, and on **start+inc+inc** the third and so on.

For further information on banking, see *3.13 Banking*.

-DSYMB=value

Sets SYMB to **value**

The **-D** switch allows definition of absolute symbols in the command line, i.e. entry points. This is particularly suited for configuration purposes. Any number of symbols can be defined, although it may require the use of the **-f** switch (extended command line). The symbol(s) will belong to the linker generated module *?ABS_ENTRY_MOD*, for references.

-Z(type)segment(s)

Define segments.

The syntax is as follows:

-Z[<(*type*)>]<*segment-list*>[<*op*><*range-list*>]

(type) is optional (see *3.5 Define Segment*)

segment-list is a list of segments with a comma as delimiter. The segments will be allocated "in order of appearance", i.e. if "CODE2" is preceded by "CODE1" then the segment "CODE1" will be first. It is possible to increase the amount of memory reserved for a segment. This is performed by appending "+*nn*" to the segments name, i.e "CODE+100" will increase the space reserved by 100 (hexadecimal).

op can either be "=" or "#". This will determine the "way of allocation", i.e. if the segments are to be allocated upwards (=) or downwards (#). If the segments are to be allocated downwards (#), then the "first" segment will have the highest address, next the second highest address and so forth. **op** is optional.

range is a list of addresses (just a number) and/or ranges ("*number*"-"*number*"), separated by comma. If the number is preceded by a dot it is considered to be a decimal value, otherwise a hexadecimal. This will define the areas where the segments will be placed. **range** is optional.

A short example:

-Z(untyped)SEGA,SEGB=100-200,400-700,1000

The segments SEGA and SEGB will be of type UNTYPED. The segment SEGA will be placed between address 100 and 200 if it fits, if not, the range 400 to 700 may be large enough, otherwise it will start on 1000. The segment SEGB will be placed, according to the same rules, *after* segment SEGA, that is, if SEGA fits in 100 to 200 then SEGB will be put there, if there is enough space left. Otherwise SEGB will go to the 400 to 700 range if large enough, or else it will start at 1000. or further information see next section.

3.5 Define Segment

The complete specification of the define segment option (-Z) is:

-z<(type)><segment list>	alloc. depends on first seg
-z<(type)><segment list>=<mem def>	upwards allocation.
-z<(type)><segment list>#<mem def>	downwards allocation.

The *(type)* defines the type of which *all* of the segments in the segment definition are. The different types are:

CODE	For code
DATA	For data
UNTYPED	Default if *"(type)"* is not specified.
XDATA	Extern data
IDATA	Internal data
BIT	Bit data

CODE, DATA, UNTYPED may be used in the formats *extended-tekhex*, *aomf8051*, *ashling-6301*, *ashling-6801*, *ashling-8080*, *ashling-8085*, *ashling-z80*, *ashling-64180*, *hp-symb*, *nec-symb* and *nec*. XDATA, IDATA and BIT may only be used in the *aomf8051* format.

The *segment list* is a list of segments on the format:

<+inc>,<+inc>,...

inc (optional) is an increment of the segment length. The increment may be given in decimals by using a dot (.) before the number, otherwise it is hex representation.

NOTE: This list also defines the order in which the segments are dumped. For cpu 8086, an additional option exists, by using ':' instead of ',' as a delimiter, the segments are defined as *COMBINED*.

mem def is one or more memory areas, and/or an address, where the segment(s) are dumped. *mem def* may be defined as follows:

address-address
: the code will be dumped between these addresses, up or down depending on the way of allocation.

address
: the code will be dumped from this address and up (or down, if that type of allocation is chosen).

Examples

- Allocation upwards -

SEGA,SEGB+10,C+.10=10-.50,40

This means: Increase SEGB by 10 (hex) and increase C by 10 (dec). Now, if SEGA fits in the memory area 10 (hex) to 50 (dec), it is put there, from 10 (hex) and upwards against 50 (dec) otherwise it is dumped from 40 (hex) and upwards. Then the linker tries to fit SEGB and C in, SEGB first and then C. SEGA will start at the lowest address, then SEGB and last, C (at the highest address - compared to SEGA and SEGB).

- Allocation downwards -

SEGA,SEGB,C#.10-.50,.90

This means: If SEGA fits in the memory area 10 (dec) to 50 (dec), It is put from 50 (dec) and down towards 10 (dec), otherwise the segment is dumped from 90 (dec) and down (if this range is too short, it will cause the error "Segment *segment* overlap segment *segment*"). Then the linker tries to fit SEGB and C in, SEGB first and then C. SEGA will start at the highest address, then SEGB and last, C (at the lowest address - compared to SEGA and SEGB).

- Priority allocation -

SEGA,SEGB+10,C+.10 SEGB,DOD=1000

This means: Increase SEGB by 10 (hex) and increase C by 10 (dec). Then, dump SEGA first, then, from 1000 (hex) SEGB and DOD, and last, at the highest address, segment C.

NOTE. You can create your own code segments by using the compiler's -R option (review the C-chapter). The best way to create your own data segment is to use the librarian to rename a data segment and insert this new data segment in the linker command file. (Also, review the technique shown in section 1.14 Configuration Issues in the C-chapter).

3.6 File Extensions

Input and output (i.e. object and hex) files have CPU or format dependent file extensions.

File extension of the *List file* is lst, and xcl of the *Command file.*

In all XLINK file specifiers, the default file extension can always be overridden.

3.7 Segments and Allocation

Segments are of three kinds: *Relative, common* and *stack.* The *stack* segment, by default, grows from high to low addresses. *Relative* and *common* are, by default, allocated from low to high. There are two ways to override the default allocation:

1). The segment is defined (see option -Z) to be located upwards (=) or downwards (#). Example:

-ZSTACK#1000 The segment STACK will be allocated from 1000H and downwards.

2). If a mix of *relative* or *common,* and *stack* segments occur in a segment definition (option -Z) the linker will produce a warning, but allocate the segments according to the default allocation set by the *first* segment in the segment definition. Example:

-ZCSTART,STACK If CSTART is *relative* then STACK will also be allocated upwards, i.e from low to high, regardless of what type it is.

Each undefined segment, generates a warning, but they will be allocated as if they were defined together with the 'last' segment, i.e the (last defined) segment at the highest address.

3.8 Command Files

The "-f" option.

The command file is just an extended command line, the syntax is exactly the same as for the command line, the only difference is that *end of line* is treated as a space (i.e a delimiter) while *end of file* terminates the extended command line. If a command file is used, optional parameters may be added after the file name. Note that the -! option used to increase readability in complex command files.

Example:

XLINK **-f comfile main function -K**

This command causes the files *main and function* to be linked, with the options and files specified in the command file, without loading the default libraries (option -K).

Review section 1.13 Linking in the C-chapter for a discussion of the supplied linker command file.

3.9 Default Libraries

Every loaded module will set one of 256 flag bits, where each bit represents a LAN value. XLINK will (until no unresolved externals exist or no flag bits are set) try to open and make a standard load operation on files with name DFLTLB##.r??, where "##" is the hexadecimal representation of the set flag bit, and the r?? is the standard CPU dependent extension. If XLINK fails to open the DFLTLB##.r?? file, the flag bit is just reset (i.e. this is not an error condition). It is possible to suppress this loading with the '-K' option.

Path to a default library can be set through an *environment variable*, XLINK_DFLTDIR. Note that before a DFLTLB file is loaded, the associated flag bit (e.g. 22 for DFLTLB16) is reset, but that referenced library modules also set the appropriate flag bits. That is, the process can be recursive.

3.10 Environment Variables

The linker uses seven *environment variables* which can be set in the host environment:

XLINK_DFLTDIR	: Sets a path to a default directory (for DFLTLB??.r??).
XLINK_PAGE	: The number of lines per page (20 - 150).
XLINK_COLUMNS	: The number of columns per line, of the list file, default is 80 columns.
XLINK_CPU	: Default cpu.
XLINK_FORMAT	: Default output format.
XLINK_MEMORY	: If this variable is zero (0), then the linker is file bound, else it is memory bound (see *3.12 File bound processing*).
XLINK_ENVPAR	: Extension of command line. Can hold command line data in the normal format.

Except for **XLINK_ENVPAR** parameter the environment variable sets default values which can be overruled by the corresponding value given on the command line. (I.e. -c8051 supersede a set **XLINK_CPU** environment variable).

3.11 Linker Diagnostics

The error messages produced by the linker falls into five categories:

1. Linker warning messages
2. Linker error messages
3. Linker fatal error messages
4. Memory overflow message
5. Linker internal errors

Linker Warning Messages

Linker warning messages will appear when the linker has found something that may be wrong, but the code produced can still be correct. Also, review Appendix F.

Linker Error Messages

Linker error messages are produced when the linker has found something wrong, it will not abort the linking process, but the code produced is probably faulty. Also, review Appendix F.

Linker Fatal Errors

Linker fatal error messages aborts the linking process, and occurs when it is useless to continue the linking, that is, the fault is irrecoverable. Also, review Appendix F.

Memory Overflow Message

The Archimedes linker is a memory-based linker. As such, in case of a system with a small primary memory or very large source files, it may run out of memory. This is recognized by the special message:

```
***LINKER OUT OF MEMORY***
      Dynamic memory used: nnnnnn bytes
```

If such a situation occurs the cure is either to add system memory or to use the option '-m' (enable file bound processing, see *3.12 File bound processing*). However, with 512K RAM the linker capacity is sufficient for most reasonably sized object files.

Linker Internal Errors

During linking a number of internal consistency checks are performed and if any of these checks fail the linker will terminate after giving a short description of the problem. Such errors should normally not occur and should be reported to Archimedes technical support group. Please do not forget to include all possible information about the problem and preferably also a diskette with the program that generated the internal error.

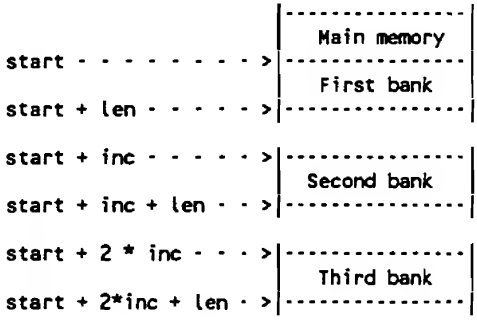
3.12 File Bound Processing

File bound processing means that instead of reading all the files into RAM memory, the linker uses file pointers to all segments and modules in the original files. This will considerably reduce the amount of main memory used for the linking process. The time needed to link, with file bound processing may vary somewhat, but is longer on MS-DOS, and shorter on VAX/VMS and VAX/UNIX.

3.13 Banking

When 8-bits and 16-bits cpu's are used the address space may be too small. This may be solved through the use of bank switching, which is featured in Archimedes Software's compilers.

When the banking option (-b) is used, the linker will place the segments in banks. Packing of the segments are dependent on the delimiter between the segments in the bank list. If it is a colon, the segment after it will start in a new bank. If it is a comma, the segment will start in the bank holding the previous segment. A schematic picture of bank switching:



Note that when linking bank segments the range check and the overlay check is automatically turned off. For more information on bank switching review the C-chapter.

3.14 Cross Reference List

When using one of the map commands, a list of symbols, types and values is produced. First there is a header (only if -l option was issued), containing the target cpu output file(s) and the actual command line parameters. Then comes the module and segment maps and their parameters, and last, if the "i" option is used, an index list to help locating entries, segments etc.

Cross-reference list

```
#########################################################################
#                                                                       #
#    Archimedes Linker          V3.00 A/VU2       08/Jul/87  09:58:01  #
#                                                                       #
#         Target cpu    ="cpu name"                                     #
#         List file     ="list file name"                               #
#         Output file 1 ="output file name"                             #
#         Output file 2 ="output file name"                             #
#         Object format ="object format"                                #
#         Command line  ="command line"                                 #
#                                                                       #
#                        (c) Copyright Archimedes Software, Inc. 1987 #
#########################################################################


                - Cross-reference -                                page
2


          ************************************
          *                                  *
          *        CROSS REFERENCE           *
          *                                  *
          ************************************
```

```
  Program entry at : "address" in module :"module name"


            **********************************
            *                                *
            *          MODULE MAP            *
            *                                *
            **********************************

  FILE NAME : "file name"
  "type of module (p2)", NAME : "module name"
     ABSOLUTE ENTRIES             ADDRESS            REF BY MODULE
     ================             =======            =============
     "absolute entry name"        "its address"      "module name"
  ...

     ABSOLUTE LOCALS              ADDRESS
     ===============              =======
     "absolute local name"        "its address"
  ...

  SEGMENTS IN THE MODULE
  ======================
"segment name"
  "segment type (p3)", address : "start address" - "end address"
    ENTRIES                       ADDRESS            REF BY MODULE
    "reloc entry name (p4)"       "its address"      "module name"
    ...


    LOCALS                        ADDRESS
    "reloc local name (p5)"       "its address"
    ...

   ------------------------------------------------------------------------
"segment name"
...

            **********************************
            *                                *
            *     SEGMENTS IN DUMP ORDER     *
            *                                *
            **********************************
```

```
  SEGMENT        START ADDRESS    END ADDRESS    TYPE    ORG     P/N     ALIGN
S86
  ========       =============    ===========    ====    ===     ===     =====
===
"segment name"   "start adr"      "end adr"      (p6)    (p7)    (p8)     (p9)
(p10)
                 "start adr"      "end adr"      (p6)    (p7)    (p8)     (p9)
(p10)
                 ...
                   (If the segments are banked, then the total layout of
how
                     the segments are allocated in the banks, is shown.)
...
            ***********************************
            *                                 *
            *    INDEX FOR CROSS REFERENCE    *
            *                                 *
            ***********************************

"name"
     "what it is (p11)" : "on which page(s) it may be found"
...

            ***********************************
            *                                 *
            *     END OF CROSS REFERENCE      *
            *                                 *
            ***********************************

Warnings: none
Errors: none
```

Explanation of table items

p1: Types of program entries are *absolute* or *relocatable*.
p2: Types of modules are either *program* or *library*.
p3: Types of segment are *relative*, *stack*, *banked* or *common*.
p4: The entries are relocatable with its segment.
p5: The locals are relocatable with its segment.
p6: Type of segment: *relative* (*rel*), *stack* (*stc*), *banked* (*bnk*) or *common* (*com*).
p7: Origin. I.e. if the segments start address is *absolute* (*stc*) or *floating* (*flt*).
p8: Positive/Negative. I.e. if the segment is allocated upwards (*pos*) or downwards (*neg*) in memory.
p9: Alignment. 2**align = maximum alignment factor found.
p10: Open (opn) or closed (cis) segment, *FOR CPU 8086 ONLY*.
p11: The name can be an *entry*,*external*,*local*,*module* or a *segment*.

3.15 Output Formats

XLINK supports a number of popular ASCII and some binary formats , currently used by many monitor programs (debuggers), PROM programmers and emulators. Each CPU has a default output format. The default output format can be overridden by specifying wanted format with the -F option. On the following pages there is a list of the available formats and a short description of most of them:

FORMAT	FILE FORMAT	DEFAULT EXT
AOMF8051	binary	from CPU
AOMF8096	binary	from CPU
ASHLING	binary	none
ASHLING-6301	binary	from CPU
ASHLING-64180	binary	from CPU
ASHLING-6801	binary	from CPU
ASHLING-8080	binary	from CPU
ASHLING-8085	binary	from CPU
ASHLING-z80	binary	from CPU
EXTENDED-TEKHEX	ASCII	from CPU
HP-CODE	binary	x
HP-SYMB	binary	l
INTEL-STANDARD	ASCII	from CPU
INTEL-EXTENDED	ASCII	from CPU
MILLENIUM (Tektronix)	ASCII	from CPU
MOTOROLA	ASCII	from CPU
MPDS-CODE	binary	tsk
MPDS-SYMB	binary	sym
MSD	ASCII	sym
NEC-SYMBOLIC	ASCII	sym
PENTICA-A	ASCII	sym
RCA	ASCII	from CPU
SYMBOLIC	ASCII	from CPU
TEKTRONIX (Millenium)	ASCII	hex
TEXAS-INSTRUMENT (TMS7000)	ASCII	from CPU
TYPED	ASCII	from CPU

Then there are nine other formats, which generates *two* different output formats.

FORMAT	First Format	def ext	Second Format	def ext
HP	HP-CODE	x	HP-SYMB	l
MPDS	MPDS-CODE	tsk	MPDS-SYMB	sym
MPDS-I	Intel-std	hex	MPDS-SYMB	sym
MPDS-M	Motorola	s19	MPDS-SYMB	sym
MSD-I	Intel-std	hex	MSD	sym
MSD-M	Motorola	hex	MSD	sym
MSD-T	Millenium	hex	MSD	sym
NEC	Intel-std	hex	NEC-SYMB	sym
PENTICA-I	Intel-std	obj	Pentica-a	sym
PENTICA-M	Motorola	obj	Pentica-a	sym

3.15.1 Output formats - Aomf8051, Aomf8096 or Ashling

Aomf8051, Aomf8096 or Ashling

For explanations see the EXTERNAL PRODUCT SPECIFICATION FOR THE 8051 OBJECT MODULE FORMATS rev V5.0, EXTERNAL PRODUCT SPECIFICATION FOR THE MCS-96 OBJECT MODULE FORMAT X142 rev A, Ashling Microsystems 6301 Object Module Format Ver 1.0.1 or Ashling Microsystems 64180 Object Module Format Ver 1.0.1.

3.15.2 Output formats - Extended-Tekhex

EXTENDED-TEKHEX

Everything is in hexadecimal representation, i.e this format is a hex dump. (*abs = absolutes*).

<Segment record><Public abs><Local abs><Code><End>

<*Segment record*> (repeatable):

The record contains two records, both repeatable:

<*Public relative symbols*><*Local relative symbols*>

<*Public relative symbols*> (repeatable):
%<*Block length*>3<*CHK*><*Segment definition*>
<*Symbol definition*><*eoln*>

<*Local relative symbols*> (repeatable) :
%<*Block length*>3<*CHK*><*Segment definition*>
<*Symbol definition*><*eoln*>

<*Public absolutes*> (repeatable):

%<*Block length*>3<*CHK*>e%_ABSOLUTE_VAR00x0x
<*Symbol definition*><*eoln*>

<*Local absolutes*> (repeatable):

%<*Block length*>3<*CHK*>e%_ABSOLUTE_VAR00x0x
<*Symbol definition*><*eoln*>

<*Code*> (repeatable):

The *Code* can be preceded by a *Type* declaration, which defines of which type *Code* are (*Type* is 4, for code, or 8, for data). *NOTE:* All *Code* will be of the defined type, until a *Type* record changes the type.

%<*Block length*>5<*CHK*><*Type*>
%<*Block length*>6<*CHK*><*Load address*><*Data*><*eoln*>

<*End*>:
%<*Block length*>6<*CHK*><*Program entry*><*eoln*>

Explanation of records

<Block length>

Length of block, excluding '%' and *<eoln>*.

<CHK>

Checksum.

<Segment definition>

1. One byte, the length of the name, max 16.
2. The name, No more than 16 characters.
3. '0'.
4. One byte, the length of the base address.
5. Base address (hexadecimal).
6. One byte, the length of the segment length.
7. The segment length (hexadecimal).

<Symbol definition>

This record is repeatable and contains:

1. One byte, type of symbol, (1-8).
2. One byte, symbol name length (hex).
3. Symbol name.
4. One byte (hex), value length.
5. Value in hexadecimal representation.

x

One (1) to eight (8) zeros, depending on cpu.

<Load address>

1. One byte, the load address length.
2. The load address.

<Data>

Repeatable.Data bytes in hex representation.

<Program entry>

1. One byte. The program entry length.
2. The program entry, hex repr.

3.15.3 Output formats - HP-Code and HP-Symbol

HP-CODE *AND HP-SYMBOL*

For explanations see the HP manual: *Hosted Development System*, chapter *absolute file format* and *linker symbol file format.*

3.15.4 Output formats - Intel

INTEL-STANDARD, INTEL-EXTENDED

:nnoooo00ddddddcc	Data record (nn=01 to 10) (dd=data, cc=checksum, oooo=offset/address)
:01000002sssscc	INTEL-EXTENDED: 8086 type segment record (ssss=segment)
:01000003ssssoooocc	INTEL-EXTENDED: 8086 type program entry record
:00oooo01FF	End of file record (Last) (oooo=program entry address)

cc is the checksum, and the sum of all bytes (MOD 256) including the checksum itself, should be zero.

3.15.5 Output formats - Millenium

MILLENIUM (Tektronix)

/aaaannccdddddddee	Data record (nn=01 to 10)
/00000000	End of file record (Last)

aaaa	Load address.
nn	The number of data bytes.
cc	Checksum, formed by adding all digits a and n MOD 256.
dd	Data bytes.
ee	Checksum, formed by adding all digits d MOD 256.

3.15.6 Output formats - Motorola

MOTOROLA

S0nnppppxxxxxxxxcc	First record in file (address pppp=0000) (with the name of the first module).
S1nnppppddddddddcc	Data record with a 16-bit address.
S2nnpppppppddddddddcc	Data record with a 24-bit address.
S8nneeeeeeecc	End of file record (nn=04) (with 24-bit program entry).
S9nneeeeecc	End of file record (nn=03) (with 16-bit program entry).

nn	Number of bytes (xx+pp+dd+cc) in record.
pppp and pppppp	Load addresses.
dd	Data bytes, 1 to 16.
xxxxxxxx	The name of the first module (coded with two hex digits for every character).
eeee and eeeeee	The program entry (execution) addresses.
cc	Checksum :

cc:=(255-(SUM(pp)+SUM(dd)+SUM(xx)+SUM(ee)+nn)) MOD 256.

3.15.7 Output formats - MPDS-CODE

MPDS-CODE

This is just a binary hex dump, i.e. the code is dumped in sequential order from address zero to the highest address, with zeros between the code sections.

MPDS-SYMB

Each module have a *Module record*:
FE<*Module name length*><*Module name*>
<*Record length*><*constant*><*Symbol record*>

The *Symbol record* consists of a variable number of:
<*Symbol name length*><*Symbol name*><*Symbol value*>

Module and *Symbol name length* is one byte long.
Module and *Symbol name* is *length* long.
Record length (*three bytes*) is the length of the rest of the block.
constant is two (2) or three (3), depending on how many bytes the symbol value is.
Symbol value is two or three bytes long.

3.15.8 Output formats - MSD

MSD

<*Value*> P <*Symbol name*><*End of line*>
Symbol declaration. Value is two hex bytes.
Symbol name is the name of the entry or the local.

3.15.9 Output formats - NEC-SYMBOLIC

NEC-SYMBOLIC

	2 char	4 char	max 8 char
Symbol Table Start	# 04		
	; FF	4 Blanks (spaces)	Module Name
	Type	Symbol Value	Public Symbol Name
	Type	Symbol Value	Public Symbol Name
	.	.	.
	.	.	.
Local Symbol Start	< Type	Symbol Value	Local Symbol Name
	Type	Symbol Value	Local Symbol Name
	.	.	.
	.	.	.
Repeat Begins	; FF	4 Blanks (spaces)	Module Name
	Type	Symbol Value	Public Symbol Name
Symbol Table End	=		

NOTE: Each row above ends with carriage return *and* line feed

Value	*Attribute*
00	*Number*
01	*Code*
02	*Data*
05	*Bit*
FF	*Module Name*

3.15.10 Output formats - Pentica-a

PENTICA-A

BEGIN<*End of line*> File header.

<*Value*> P <*Symbol name*><*End of line*>
Symbol declaration. Value is two hex bytes. Symbol name is the name of the entry or the local.

END<*End of line*> File trailer.

3.15.11 Output formats - RCA

RCA

48 null characters	Header
!Maaaa ddddddddddd;	First record
aaaa dddddddd;	Data record (<=16 bytes)
aaaa dddddddd	Last data record (<=16 bytes)
48 null characters	Trailer

where:

aaaa	Load address.
dd	Data bytes.

3.15.12 Output formats - Symbolic & Typed

SYMBOLIC, TYPED

<I> 0 <SYMBOL> <CHK>	Start of a new module
<I> 1 <LA> <DATA> <CHK>	An absolute data record in current. module.
<I> 2 <LA> <DATA> <CHK>	A relocatable data record in current module.
<I> 3 <VAL> <SYMBOL> <TYPE>* <CHK>	An absolute local in curr. mod.
<I> 4 <VAL> <SYMBOL> <TYPE>* <CHK>	A relocatable local in current module.
<I> 5 <VAL> <SYMBOL> <TYPE>* <CHK>	An absolute global in curr. mod.
<I> 6 <VAL> <SYMBOL> <TYPE>* <CHK>	A relocatable global in current module.
<I> 7 <VAL> <CHK>	An absolute program entry in current module.
<I> 8 <VAL> <CHK>	A relocatable program entry in current module.
<I> 9 <CHK>	Last record in file

* <TYPE> is only available in the TYPED format.

Record	Description
<I>	A one character identifier. % = TYPED and # = SYMBOLIC.
<LA> <VAL>	A load address or value with 4 possible variations, where the first character in the record gives the type. 16-bit values are expressed as 4 hex ASCII digits and 32-bit as 8. For other processors than 8086, only type 0 and 1 are used. 0 <16-bit address> 1 <32-bit address> 2 <16-bit 8086-type para. no.> <16-bit offset> 3 <16-bit 8086-type para. no.> <32-bit offset>
<DATA>	A variable length record, consisting of a 2-digit hex ASCII value holding the number of data bytes in the record. After the length indicator, the data bytes are output,where each byte is represented by a 2-digit hex ASCII number. An exception is relocatable 8086-type paragraph numbers, which are 16-bit words written as: $ <4 hex ASCII digits>.
<SYMBOL>	A variable length record, consisting of a 2-digit hex ASCII value holding the length of the symbol, followed by the symbol itself, with all eventual non-printable characters converted to "." (dot).
<TYPE>	A variable length record, consisting of a 2-digit hex ASCII value holding the number of bytes, in the variable part of the type record. After the length indicator, the type information is output, with each byte expressed as a 2-digit hex ASCII number.
<CHK>	A 2-digit hex ASCII value containing the modulus 256 sum, of all characters in the record (excluding <CHK>). After the <CHK> record, an end of line character is output.

3.15.13 Output formats - Texas

TEXAS-INSTR (TMS7000)

```
K0000xxxxxxxx7ccccF                    Header with name (xxxxxxxx)

9aaaa*ddBddddBddddBddddd7ccccF         Data record (1-16 bytes)

1aaaa7ccccF                            Program entry record
:                                      End of file record.
```

where:

9aaaa	Load address
Bdddd	Data word
*dd	Data byte
7cccc	Checksum
F	End of line

The checksum (cccc) is the negated sum of all characters in the line up to and including the checksum tag (7).

4.1 Librarian - Introduction

XLIB is a versatile tool for program development which has been designed to aid the maintenance and creation of relocatable libraries. Maintaining libraries is a must for the advanced users, and especially those who use compilers will appreciate the XLIB utility program, although XLIB is equally applicable to modularized assembly language programming.

XLIB is like the universal linker, XLINK, a single program that due to a standardized object format called UBROF (Universal Binary Relocatable Object Format) can support a wide range of 8-32 bit byte oriented processors (applies to almost all major microprocessors of today).

See Appendix G for a list of all Librarian error messages.

4.2 Capabilities and Uses

Now, what exactly can you do with XLIB? A list showing the possible manipulations on object files, is as follows:

1. Merge object files from different assemblies/compilations in order to create libraries.

2. Delete individual modules in a file.

3. Change (move) the order of modules in a file. This is sometimes needed because of the loading strategy used in XLINK.

4. Check/list CRC (Cyclic Redundancy Check), which is the last element of every object module.

5. Rename modules, segments, externals or entries.

6. Change the properties of a module to "Library" (conditionally loaded) or "Program" (unconditionally loaded).

7. List modules, segments, externals, entries, locals or all the UBROF record types in a special ASCII format.

In all examples user input is underlined.

4.3 Operating Instructions

There are two ways to start XLIB:

```
XLIB

XLIB  Command file  [P0,P1...P9]
```

If XLIB is not followed by any parameters, XLIB will prompt (with an '*') the user for direct command input. In the second example, commands are read from a file (see section 4.5).

4.4 Parameters, Delimiters and Commands

Commands and parameters should be separated by commas, <CR> or spaces, and prompters are issued in the absence of a parameter.

Command identifiers consist of one or more words, separated with hyphens ('-'). The individual words of an identifier may be abbreviated to the limit of ambiguity. Examples: LIST-MODULES may be written as LIST-MOD or L-M, but not as LIST-, because LIST- is ambiguous with reference to other LIST commands.

XLIB has two types of parameters, one that has no defaults (prompter is repeated until a parameter is found), and one where there exists a valid default. The parameters that do not have any defaults are marked <Parm>, while the other type is marked [<Parm>]. The parameters that have default values are:

Parameter	Default Value
[<List file>]	The user terminal
[<Start mod>]	First module of the object file
[<End mod>]	Last module of the object file

Default parameters should be written ',,', but default values will also be used when only a <CR> is issued, in response to a XLIB parameter prompt.

Commands and file names may be written in any mix of upper and lowercase letters, but object code symbols (module names etc.) must always be written in their actual form (i.e. no automatic translation is performed).

4.5 Command Files

If a command file is used, optional parameters may be added after the file name. Parameters must be separated with spaces or commas, and a default parameter can be issued by writing ',,'. Inside the command file, parameters will be substituted at every occurance of a backslash ('\') followed by a decimal digit (0-9). An example:

```
LIST-MOD  \4  PRN
```

If COMFIL contains a line like the one above, the invocation command:

```
XLIB  COMFIL,ABC  G,,78,CODE  SOME
```

would make the line be interpreted as:

```
LIST-MOD  CODE  PRN
```

Immediately after a substitute parameter, a default value may be added. The default value is recognized as a string enclosed in single quotes (e.g. 'Parm').

If the requested parameter is missing in the XLIB invocation line, or is written as ',,', the default string will replace the \n parameter.

An example of a line with a substitute parameter and a default value:

```
LIST-MOD  \4'OBJ'  PRN
```

There is also another form of parameter that can be specified in a command file, called the "interactive substitute". These parameters should be written '\?', and will make XLIB stop and wait for a user response. The interactive parameters may also have a user prompt string attached immediately after the question mark.

An example:

```
LIST-MOD  \?'YOUR FILE PLEASE: '  PRN
```

The \n parameter is replaced with the user input except for <CR>.

Note that when XLIB is used with a command file, it operates completely "silent" except when errors occur. This batch-like operation, can be further controlled by the commands ON-ERROR-EXIT, ECHO-INPUT and REMARK.

Before any command that reads or writes object files can be issued, the target 8051 CPU must be defined (with DEFINE-CPU), which sets the unique default file extension (type).

When should command files be used?

Command files can simplify the use of XLIB, when certain sequences of commands are to be frequently performed. A typical example is when you have designed a large library file, consisting of numerous object modules. After a while the specifications may change or bugs may be found causing replacement of one or more object modules. The required sequence of commands needed to update the library file (from an altered object file) can be put into a command file. In addition to repeating command sequences, additional flexibility can be achieved by using substitute parameters. A common situation calling for more flexibility is when target libraries do not reside on the same directory. Here a parameter may optionally specify the directory. An example of such a command file is:

```
DEF-CPU  8051
REPLACE-MOD  \0  \1MYLIB
LIST-DATE-STAMPS  \1MYLIB  \2'PRN'
EXIT
```

Some examples on how the command file could be used:

```
XLIB  COMFIL  ALTFIL
XLIB  COMFIL  ALTFIL  \PROJ\
XLIB  COMFIL  ALTFIL,,CON
```

4.6 Module Expressions

In most of the XLIB commands, you can or must specify a source module (like <Old name> in RENAME-MODULE), or a range of modules (<Start mod>,<End mod>).

Internally in all XLIB operations modules are numbered from one and up. The modules may be expressed as the actual name of the module, or as the name plus or minus a relative expression, or as an absolute number. The latter is very useful when a module name is very long, unknown or contains "unusual" characters (like space or comma).

Below is a list of the available variations on module expressions:

Name	The actual name of the module
<3>	The third module
<$>	The last module
<Name+4>	The module 4 modules ahead of "Name"
<Name-12>	The module 12 modules before "Name"
<$-2>	The last module-2

The command LIST-MOD FILE,,<$-2> will thus list the three last modules in FILE on the user terminal.

4.7 Command Line

Occasionally, XLIB's commands with their parameters exceed the width of the display device. In those cases, command lines may be logically extended, by terminating the lines with an ampersand ('&'), immediately before the <CR>. XLIB will then respond with an angle bracket and additional input can be given. This process may be repeated if needed, but the total length of a logical line must never exceed 512 characters. An example:

```
*RENAME-MODULE MOD <1>&
>THENEWANDVERYLONGBUTSTILLVALIDMODULENAME
*
```

Note: Command line extension also works on the XLIB start command.

Normal line editing is always possible.

4.8 List Format

When using one of the LIST commands, you will get a list of symbols, where each symbol has a prefix. The following prefixes are to be found:

```
nn.  Pgm  SYMBOL      A program module with relative number nn.
nn.  Lib  SYMBOL      A library module with relative number nn.
Ext  SYMBOL           An external in the current module.
Ent  SYMBOL           An entry in the current module.
Loc  SYMBOL           A local in the current module.
Rel  SYMBOL           A standard segment in the current module.
Stk  SYMBOL           A stack segment in the current module.
Com  SYMBOL           A common segment in the current module.
```

If a XLIB list is directed to another file than the user terminal, a header with some important data will be added to the list file:

```
#########################################################################
#                                                                       #
#     Archimedes Library Manager V1.80/MD2       01/Jan/86  22:51:50    #
#                                                                       #
#         Command =  list-modules                                       #
#         CPU     =  8051                                               #
#         File    =  mod.r03                                            #
#         Modules =  1-5(5)                                             #
#                                                                       #
#                          (c) Copyright Archimedes Software, Inc. 1985 #
#########################################################################

      1.  Lib  TTY
      2.  Lib  DISK
      3.  Pgm  DOIO
      4.  Pgm  GET
      5.  Pgm  PUT
```

4.9 File Extensions

All object files (<Object file>, <Source file> and <Dest file>) have the same 8051 CPU dependent file extensions as the rest of the package. File extension on the <List file> is LST, and XLB on the <Command file>.

In all XLIB file specifiers, the default file extension can always be overridden.

4.10 Command Modes and Sample Session

If a command is not followed by any parameters, the command processor in XLIB assumes that the "prompt mode" is requested. In this mode some kind of response (a string, ',,' or <CR>) must be given to all parameters, regardless if they have a default. This mode is preferable when you have forgotten some command parameter, because the XLIB prompters will tell you what type of parameter it is expecting.

In the other XLIB mode a command is directly followed by one or more parameters. This mode is called the "direct mode". In the direct mode default values will be automatically inserted, and if a parameter that does not have a default value is missing an error message will be issued.

```
xlib

   Archimedes Library Manager V1.80/MD2
(c) Copyright Archimedes Software, Inc. 1985.
```

This is a sample session showing how the "direct mode" differs from the "prompt mode". Before work on object files can begin, define the 8051 target CPU. Use: DEFINE-CPU <Cpu>

```
*def-cpu              % Missing parameter = "prompt mode".
Target CPU=8051       % 8051 target
```

Use *exit to return to system.

Assume we have a file called "mod", which contains valid object modules. Now, list all the modules on the terminal. Use: LIST-MODULES <Object file> [<List file>] [<Start mod>] [<End mod>]

In "prompt mode" :

```
*lis-mod                % No parameters = "prompt mode"
Source file=mod
List file=              % <CR> = User terminal
Start module=           % <CR> = First module in file
End module=             % <CR> = Last module in file

      1.  Lib  TTY
      2.  Lib  DISK
      3.  Pgm  DOIO
      4.  Pgm  GET
      5.  Pgm  PUT
```

List from module "DOIO" to the last module, on the terminal.

```
*li-mod                 % No parameters = "prompt mode"
Source file=mod
List file=
Start module=DOIO,,     % ',,' = Use default => Last module

      3.  Pgm  DOIO
      4.  Pgm  GET
      5.  Pgm  PUT
```

Note that in "prompt mode" there must be a response to all parameters. Now perform the previous operations in "direct mode". First list all the modules. Then from DOIO to last module.

```
*li-mod mod             % There was a parameter = "direct mode"
                        % The missing parameters was filled with:
      1.  Lib  TTY      %   [<List file>] => User terminal
      2.  Lib  DISK     %   [<First mod>] => First module in file
      3.  Pgm  DOIO     %   [<Last mod>]  => Last module in file
      4.  Pgm  GET
      5.  Pgm  PUT


*li-mo mod,,DOIO        % ',,' = Use default => User terminal
                          % Missing [<Last mod>] => Last module in
file
      3.  Pgm  DOIO
      4.  Pgm  GET
      5.  Pgm  PUT
```

4.11 Command summary

On the following pages there is a printout of the HELP file which contains the syntactic and functional description of all commands.

HELP [<Command>] [<List file>]

If the HELP command is followed by ',,' only a list of the available commands will be displayed on the user terminal. If instead a parameter is given, all commands which match the parameter will be displayed with a brief explanation of their syntax and function. An '*' matches all commands. HELP output can be directed to any file.

```
Defaults:  [<List file>]      =>   The user terminal
           [<Start mod>]      =>   The first module
           [<End mod>]        =>   The last module

Module expressions ( <Start mod> <End mod> ):
  Name              The actual name of the module
  <3>               The third module
  <$>               The last module
  <Name+4>          The module 4 modules ahead of "Name"
  <Name-12>         The module 12 modules before "Name"
  <$-2>             The last module-2
```

EXIT

Return to OS (DOS, VMS, UNIX).

REMARK <Anything>

Just a comment (to increase readability).

LIST-OBJECT-CODE <Object file> [<list file>]

List the contents of the <Object file> on the [<list file>] in a special ASCII format. (Intended only for special internal checks of software kit).

LIST-ALL-SYMBOLS <Object file> [<List file>] [<Start mod>] [<End mod>]

List from [<Start mod>] to [<End mod>], all symbols (Module names, Segments, Externals, Entries and Locals) of the <Object file> on the [<List file>].

LIST-CRC <Object file> [<List file>] [<Start mod>] [<End mod>]
List from [<Start mod>] to [<End mod>], the modules names and their associated CRC:s.

LIST-MODULES <Object file> [<List file>] [<Start mod>] [<End mod>]
List from [<Start mod>] to [<End mod>], the module names only.

LIST-ENTRIES <Object file> [<List file>] [<Start mod>] [<End mod>]
List from [<Start mod>] to [<End mod>], module names and their associated entries.

LIST-EXTERNALS <Object file> [<List file>] [<Start mod>] [<End mod>]
List from [<Start mod>] to [<End mod>], module names and their associated externals.

LIST-SEGMENTS <Object file> [<List file>] [<Start mod>] [<End mod>]
List from [<Start mod>] to [<End mod>], module names and their associated segments.

LIST-DATE-STAMPS <Object file> [<List file>] [<Start mod>] [<End mod>]
List from [<Start mod>] to [<End mod>], module names and their associated generation date.

RENAME-MODULE <Object file> <Old name> <New name>
Rename a module. Note: If there are more than one module with <Old name>, only the first encountered is changed.

RENAME-ENTRY <Object file> <Old name> <New name> [<Start mod>] [<End mod>]
Rename from [<Start mod>] to [<End mod>], all occurances of an entry with <Old name> to <New name>.

RENAME-EXTERNAL <Object file> <Old name> <New name> [<Start mod>] [<End mod>]

Rename from [<Start mod>] to [<End mod>], all occurances of an external with <Old name> to <New name>.

RENAME-GLOBAL <Object file> <Old name> <New name> [<Start mod>] [<End mod>]

Rename from [<Start mod>] to [<End mod>], all occurances of an external or entry with <Old name> to <New name>.

RENAME-SEGMENT <Object file> <Old name> <New name> [<Start mod>] [<End mod>]

Rename from [<Start mod>] to [<End mod>], all occurances of a segment with <Old name> to <New name>.

DELETE-MODULES <Object file> <Start mod> <End mod>

Delete modules from <Start mod> to <End mod>.

INSERT-MODULES <Object file> <Start mod> <End mod> <Before/After> <Dest mod>

Move the modules <Start mod> to <End mod> before or after the <Dest mod>.

REPLACE-MODULES <Source File> <Dest file>

Replace modules with the same name from <Source file> to <Dest file>. All replacements are logged on the user terminal. The main application for this command is to update large run time libraries etc.

FETCH-MODULES <Source file> <Dest file> [<Start mod>] [<End mod>]

Append [<Start mod>] to [<End mod>] of <Source file> to <Dest file>. If <Dest file> already exists, it must be empty or contain valid object modules. If <Dest file> does not exist before, it will be created.

MAKE-LIBRARY <Object file> [<Start mod>] [<End mod>]
Change from [<Start mod>] to [<End mod>], module header attributes to "conditionally loaded".

MAKE-PROGRAM <Object file> [<Start mod>] [<End mod>]
Change from [<Start mod>] to [<End mod>], module header attributes to "unconditionally loaded".

COMPACT-FILE <Object File>
Concatenates short absolute records into longer records of variable length. This is useful for library files which due to the decreased size will take up less time during the loader/linker process.

DEFINE-CPU <8051>
Must be issued before any operations on object files can be done.

ECHO-INPUT
This command is useful for debugging command files because it makes all command input visible on the user terminal also in batch mode. In the interactive mode it has no effect.

ON-ERROR-EXIT
This command makes XLIB abort if an error is found. Most suited for use in batch mode.

DIRECTORY
DIRECTORY <Directory specifier>
Display on the user terminal, all files of the type that applies to the target processor. If no <Directory specifier> is given, current directory is listed.

APPENDICES:

A. Programming Hints

B. Libraries

C. Proliferation Chip Support

D. C Error Messages

E. Assembler Error Messages

F. Linker Error Messages

G. Librarian Error Messages

H. Development Tools Support

I. Code Example

J. Memory Maps

Appendix A: Programming Hints

General C programming.
The code example in Appendix I illustrates many typical and useful constructs of the Archimedes C software.

Pointers, "input" and "output" functions.
Study section 1.17 8051 Specific Library Functions.

C compatibility with other C-compilers.
Review section 1.23 C Compatibility.

Printf, sprintf does not work properly.
You need to modify these for your specific hardware environment. Review **putchar.s03** and **getchar.c**.

C compiler options do not work properly.
CASE is significant in compiler options.

Modified cstartup routine does not work properly.
You need to define DEFMM.INC properly. Review section 1.6 Overview Memory Models.

View C source code in emulator debugging session.
Display the C source code mixed with assembly source (generated by the C compiler -q and -L switches) in the window of memory-resident utility like Side-Kick on top of the screen when you run the emulator.

Host System Testing.

In order to speed the development of a project, the majority of the "front end" code can be tested on the host system (an IBM AT) using a native compiler, linker and debugger (Microsoft C with Codeview, Borland Turbo C, or other ANSI-compatible compiler). This allows verification of the program's logic, data structures, user interface and most error handling. The main program module is shown below:

```
#include <stdio.h>
#define BUF_SIZ
#define T_SIZE 10
#define T_NUM 8
struct opt    /* setup options */
{  char baud;
   char data;
   char parity;
   char trig[T_NUM][T_SIZE];  /* trigger words */
};
extern struct opt saved_opt; /* saved options */
struct opt cur_opt; /* options, cleared at reset */
char buf[BUF_SIZ];  /* data buffer */
char *in_ptr = buf;
char *out_ptr = buf;
int count;
enum {SETUP='1', START, DUMP, XMIT, RESET};
void main()
{ cur_opt = saved_opt; /* get saved options */
  init_sio(cur_opt);   /* initialize serial port */
  while(1)
  { printf("\n\n1=Setup 2=Start 3=Dump 4=Xmit 5=Reset");
    printf("\nCommand (1-5)? ");
    switch(getchar())
    { case SETUP:
        if (setup() != 0)  /* aborted by user? */
          init_sio(cur_opt);
        break;
```

```
      case START:
        printf("\n\nStop on trigger # (1-8)? ");
        monitor(getchar());
        break;
      case DUMP:      /* examine the data buffer */
        display_buf();
        break;
      case XMIT:      /* xmit a test message */
        xmit_test();
        break;
      case RESET:     /* reset */
        in_ptr = buf;
        out_ptr = buf;
        count = 0;
        break;
      default:
        break;
    }
  }
}
```

The main() module shown above would be coded and tested first on the host system using "stub" routines for functions that have not yet been written (stubs are dummy functions that simply print their parameter lists or pass test data and return). Functions that can only be tested on the 8031 target, such as the routine init_sio() which initializes the 8031's serial port, can be conditionally compiled with the "real" or a stub function as shown below:

```
#ifdef HOST_DEBUG
void init_sio(struct opt i_opt) /* debug on host */
{
  printf("\ninit_sio: %c,%c,%c\n",i_opt.baud,
    i_opt.data,i_opt.parity);
}
#else
void init_sio(struct opt i_opt)  /* 8031 target */
{
  /* real C code to init serial port... */
}
#endif
```

When this module is compiled for the host environment (PC/AT or VAX) the symbol HOST_DEBUG should be defined on the command line to cause the init_sio() stub function to be included instead of the real one:

```
cc datamon -DHOST_DEBUG
```

Standard "C" input/output calls such as printf() and getchar() are used to display menus on the host system's screen and get keyboard input. In the case of our real target board, messages would be displayed on the LCD and input received on the hex keypad.

This portability is dependent on the implementation of the C-51 versions of putchar() and getchar(). The C-51 version of putchar() (listing not included here) would consist of a driver for the LCD that writes characters to the PORT-1 interface, scrolls or clears the display when full and handles other supervisory functions. Since printf() simply calls putchar() for character-level output, there is usually no need to modify it, though the "C" source is provided for more sophisticated display routines. The getchar() routine scans the keypad by reading PORT-3 and converts the values to ASCII characters 0 - 9. Putchar() and getchar() can be implemented in either "C" or assembler (skeletal assembler source is provided with C-51).

The function monitor(), invoked by the START menu option, enables the serial port interrupt and then displays the incoming characters pulled from the input ring buffer. If a trigger pattern has been selected, the incoming stream is checked against it and the user alerted if a match is found. The input buffer is filled by the serial port interrupt handler (described later) as characters are received. For host-system debugging, this function can be tested by filling the input buffer with test data from a disk file before entering the body of the function:

```
void monitor(unsigned char trig_no)
{
  char trig_flg;
```

```
#ifdef HOST_DEBUG  /* fills buffer test file */
  in_ptr = buf;
  count = 0;
  while(((c = getc(file_ptr)) != EOF) &&
    (count++ != BUF_SIZ))
    buf[in_ptr] = c;
#endif
  printf("\n\n1-HEX 2-ASCII");
  printf("\nDisplay type (1-2)? ");
  if(((c = getchar()) < '1') || (c > '2'))
    return();
    /* kbhit() checks for keypress */
  while(trig_flg == 0 && kbhit() == 0)
  { set_bit(ES_bit);   /* enable sio interrupts */
    if(count > 0)
    { if(c = '1')
        printf("%02X ",*out_ptr);
      else
        printf("%c",*out_ptr);
      clear_bit(ES_bit);  /* disable sio interrupts */
      count--;
      if(out_ptr++ == buf + BUF_SIZ)
        out_ptr = buf;    /* ring buffer */
      set_bit(ES_bit);    /* enable ints again */
      if(trig_compare(trig_no - '0')
      { printf("\n\nTrigger pattern #%c found.",trig_no);
        trig_flag = 1;
      }
    }
  }
```

Using Bit Manipulation Functions

The functions set_bit() and clear_bit() used above to enable and disable serial port interrupts are extensions to the "C" language that generate in-line code for accessing bit-addressable 8051 registers and internal RAM. These functions allow exploiting the bit-oriented I/O and boolean processing capabilities of the 8051 without resorting to assembler. Since these are processor-specific extensions, the C-51 compiler "-e" command line option must be used to enable them. The following code can be used to simulate these functions for host system debugging:

```
#ifdef HOST_DEBUG
char ES_bit;
char P3_5_bit;      /* P3.5 (bit 5 of port 3) */
char variable1_bit;
char variable2_bit;
/* other pseudo-bit registers and variables... */

#define set_bit(x) x=1
#define clear_bit(x) x=0
#define read_bit(x) x
#define read_bit_and_clear(x) ((x))?(x=0,1):0)
#else
#define variable1_bit 0   /*  20H bit 0         */
#define variable2_bit 1   /*      bit 1, etc....*/
/* other 8051 bit variables... */
#endif
```

When the bit function macros are expanded they will read, write and test character variables instead of the 8051 bit-addressable registers and variables. This allows the boolean expression

```
if(read_bit(variable1_bit) || read_bit(variable2_bit))
  clear_bit(P3_5_bit);
```

to expand on the host system as

```
if(variable1 || variable2)  /* if HOST_DEBUG defined */
  P3_5_bit = 0;
```

The pseudo-bit variables can be examined and manipulated using the host system debugger (e.g. CodeView) to simulate the 8051 target environment and test the logic of boolean expressions without running the code on the target.

Incidentally, the code shown above compiles to just 3 8051 instructions (8 bytes) and executes in under 5 microseconds with a 12 Mhz clock. If character, instead of bit variables are used, 10 instructions are generated and the code takes almost 4 times as long to execute.

Appendix B: Libraries.

The Archimedes C-compiler kit includes most of the important C-library functions that apply to microcontrollers, i.e. PROM-based embedded systems. The libraries for the large and medium memory models are in the **cl8051.r03** file. The corresponding small memory model c-library functions are in **cl8051s.r03** file and the banked library functions are in **cl8051b.r03.**

Before using any of these library functions you must include the appropriate header for that function in your C program. The header file (e.g., **#include <ctype.h>**) must be declared before any reference to the functions or objects it declares, or to types or macros it defines.

The floating point library contains the most popular advanced math functions. Review the header **math.h**.

The C-compiler also has special C-run time libraries (?xxxx functions). You may access some of these functions via the assembler (see section 1.11).

The following library functions are available:

CHARACTER HANDLING <ctype.h>
isalnum, isalpha, iscntrl, isdigit, islower, isprint, ispunct, isspace, isupper, tolower, toupper

NON-LOGICAL JUMPS <setjmp.h>
longjmp, setjmp

FORMATTED INPUT/OUTPUT <stdio.h>
getchar, printf, putchar, sprintf, _formatted_write

GENERAL UTILITIES <stdlib.h>
exit, calloc, free, malloc, realloc

STRING HANDLING <string.h>
strcat, strcmp, strcpy, strlen, strncat, strncmp, strncpy

MATHEMATICS <math.h>
atan, atan2, cos, exp, log, log10, modf, pow, sin, sqrt, tan

Source code is provided for all the formatted input/output library functions to allow modification for a particular target environment. All library functions above are summarized in alphabetical order below. For a detailed description you may also reference any standard C language reference book.

Summary of Library Functions

atan

double atan(double x);

Computes the arc tangent of x. Returns a value in the range
[-pi/2, pi/2].

atan2

double atan2(double y, double x);

Computes the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value. Returns a value in the range [-pi, pi].

calloc

void *calloc(size_t nmemb, size_t size);

Allocates space for an array of nmemb objects. Size of elements in bytes is specified by **size**. For details concerning changing the default heap size see the target dependent section discussing the heap. **Calloc** returns a pointer (or zero if failed to find any suitable memory) to the start (lowest byte address) of the object.

cos

double cos(double x);

Computes and returns the cosine of x (measured in radians).

exit

void exit(int status);

Normal program termination. No return to its caller.

exp

double exp(double x);

Computes and returns the exponential function of x.

free

void free(void *ptr);

Object pointed to by ptr is made available for further allocation. ptr must earlier have been assigned a value from **malloc**, **calloc**, or **realloc**. Returns no value.

getchar

int getchar(void);

(Source is included). Gets a character from standard input. Returns next character from input stream. Delivered in source format for easy adaption to the target hardware configuration.

isalnum

int isalnum(int c);

Tests for any letter or digit. Returns non-zero if test is true.

isalpha

int isalpha(int c);

Tests for any letter. Returns non-zero if test is true.

iscntrl

int iscntrl(int c);

Tests for any control character. Returns non-zero if test is true.

isdigit

int isdigit(int c);

Tests for any decimal digit. Returns non-zero if test is true.

islower

int islower(int c);

Tests for any lower-case letter. Returns non-zero if test is true.

isprint

int isprint(int c);

Tests for any printable character (including space). Returns non-zero if test is true.

ispunct

int ispunct(int c);

Tests for any printable character except space, a digit or a letter. Returns non-zero if test is true.

isspace

int isspace (int c);

Tests for the following characters: ' ' (space), '\f' (form feed), '\n' (new line), '\r' (carriage return), '\t' (horizontal tab), '\v' (vertical tab). Returns non-zero if test is true.

isupper

int isupper(int c);

Tests for any upper-case letters, returns non-zero if true.

log

double log(double x);

Computes and returns the natual logarithm of x.

log10

double log10(double x);

Computes and returns the base-ten logarithm of x.

longjmp

void longjmp(jmp_buf env, int val);

Restores what was saved by the last call to **setjmp** (corresponding jmp_buf argument).

malloc

void *malloc(size_t size);

Allocates space for an object. Its size in bytes is specified by **size**. For details concerning changing the default heap size see the target dependent section discussing the heap. **Malloc** returns a pointer (or zero if failed to find any suitable memory) to the start (lowest byte address) of the object.

modf

double modf(double value, double *iptr);

Computes the integral (stored in *iptr) and the fractional parts of value (function return). Sign of both parts is the same as for value.

pow

double pow(double x, double y);

Computes and returns x raised to the power of y.

printf

int printf(const char *format, ...);

(Source is included). Writes formatted data to standard output. Returns the number of characters transmitted or a negative value if an error was encountered. Note that output is performed through library function **putchar** which must be adapted for the actual target hardware configuration.

A summary of the most common **printf** conversion specifiers are shown below. Each conversion specification is introduced by %. Conversion specifications within brackets are optional.

%{flags} {field width} {.precision} {l} *char*

flags

- left adjusted field.

+ signed values will always begin with plus or minus sign.

space values will always begin with minus or space.

field width

Number of characters printed. If integer values are written and field width starts with a zero, values will be padded with zeros.

.precision

Number of digits for floating point value (e,E and f conversion).

l

(letter 'l') Used before d,i,u,x,X,o specifiers to denote the presence of a long integer or unsigned long value.

char

d,i	Signed decimal value.
o	Unsigned octal value.
u	Unsigned decimal value.
x	Unsigned hexadecimal (0-9, abc...) value.
X	Unsigned hexadecimal (0-9, ABC...) value.
e,E	Floating point constant: [-]d.ddde+dd.
f	Floating point constant: [-]ddd.ddd.
c	Character constant.
s	String constant.
p	Pointer value (address).
%	"%" is printed.

The "printf" and "sprintf" routines share a common formatter which is pre-installed in the library module **_frm_write**. (See description in this Appendix). This module contains a version of the formatter that does not support floating support numbers. It will complain about %e, %E or %f specifiers. If you need floating point formatting you should C-replace the original module with a (considerably larger) version which resides in file **frmwri.r03** (**frmwris.r03** small model and **frmwrib.r03** for banked model). Please note that you need to change the name of the file **frmwri*.r03** to **_frm_wri.r03** prior to replacing the common formatter. The command sequence is as follows:

```
xlib                          % The library manager
def-cpu                       % Must know the CPU
rep-mod _frm_wri cl8051       %This is the replacement
exit                          % Return to system
```

putchar

int putchar(int value);

(Source is included). Writes the character specified by value to standard output. May be implemented as a macro. Returns EOF (-1) if errors occur, otherwise the character written. This routine is delivered in source format for adaption to the target hardware configuration. Note that putchar also serves as the low-level output function in **printf**.

realloc

void *realloc(void *ptr, size_t size);

Changes the size of the object pointed to by ptr (which must have got its value from **malloc**, **calloc**, or **realloc**) to the size specified by size. Realloc returns a pointer (may be zero if no room in heap left) to the start address of the possibly moved object.

setjmp

int setjmp(jmp_buf env);

Saves its calling environment in its jmp_buf argument for later use by **longjmp**. Returns the value zero if not returned from a longjmp.

sin

double sin(double x);

Computes and returns the sine of x (measured in radians).

sprintf

int sprintf(char *s, const char *format, ...);

(Source is included). Writes formatted data to a character array as specified by s. For details concerning formatting see printf.

strcat

char *strcat(char *s1, const char *s2);

Appends a copy of the string pointed to by s2 to the end of the string pointed to by s1. Returns value of s1.

strcmp

int strcmp(const char *s1, const char *s2);

Compares the string pointed to by s1 to the string pointed to by s2. Returns an integer value greater than, equal to or less than zero if the string pointed to by s1 is greater than equal to, or less than string pointed to by s2.

strcpy

char *strcpy(char *s1, const char *s2);

Copies the string pointed to by s2 into the object pointed to by s1. Returns the value of s1.

strlen

size_t strlen(const char *s);

Calculates the number of characters in the string pointed to by s (not including NUL character). Returns the number of initial characters.

strncat

char *strncat(char *s1, const char *s2, size_t n);

Appends a copy of the string (up to n characters) pointed to by s2 to the end of the string pointed to by s1. Returns the value of s1.

strncmp

int strcmp(const char *s1, const char *s2, size_t n);

Compares the string (up to n characters) pointed to by s1 to the string pointed to by s2. Returns an integer value greater than, equal to or less than zero if the string pointed to by s1 is greater than equal to, or less than string pointed to by s2.

strncpy

char *strcpy(char *s1, const char *s2, size_t n);

Copies the string (up to n characters) pointed to by s2 into the object pointed to by s1. Returns the value of s1.

sqrt

double sqrt(double x);

Computes and returns the square root of x.

tan

double tan(double x);

Computes and returns the tangent of x (measured in radians).

tolower

int tolower(int c);

Converts an upper-case letter to the corresponding lower-case letter.

toupper

int toupper(int c);

Converts a lower-case letter to the corresponding upper-case letter.

_frm_wri

```
int _formatted_write (const char *format,
        void outputf(char, void *),
        void *secret_pointer,
        va_list ap);
```

(Source is included). This routine serves as the basic formatter for both **printf** as well as **sprintf**. Due to its universal interface, it can be adapted for user-defined purposes such as creating a routine "dprint" that writes formatted information to non-standard display device. **_frm_wri** takes as first parameter a standard **printf/sprintf** format specification (i.e. "VAL=%d\n") while the other three parameters have the following tasks:

outputf: A function pointer to a routine that actually writes a single character created by **_frm_wri**. The first parameter to this function contains the actual character value and the second a pointer which value always is equivalent to the third parameter of **_frm_wri.**

secret_pointer: This parameter is supposed to point to some type of data structure that the low-level output function may need. If there is no need for anything more than just the character value this parameter must still be specified with "(void *) 0" as well as declared in the output function.

ap: Points to the variable-argument list.

_frm_wri is reentrant (as all other C library functions) and returns an integer holding the number of characters written.

The source of **sprintf** at the next page serves as an example on how to use **_frm_wri**:

```
/*            SPRINTF.C                */

#include "stdarg.h"
#include "stdio.h"

extern int _formatted_write (const char *format,
                             void outputf(char, void *),
                             void *secret_pointer,
                             va_list ap);

static void put_c_in_string (char c, void *ptr)  /* Low-level output */
  {
    *(*(char **) ptr)++ = c;
  }

int sprintf (char *s, const char *format, ...)     /* Our main entry */
  {
    va_list ap;
    int nr_of_chars;

    va_start (ap, format);        /* Variable argument begin */
    nr_of_chars = _formatted_write (format, put_c_in_string, (void *) &s,
ap);
    va_end (ap);                  /* Variable argument end */
    *s = '\0';                    /* String should be terminated with NUL
*/
    return (nr_of_chars);         /* According to ANSI */
  }
```

Note that a number of special ANSI-defined macros residing in file **stdarg.h** must be used as in the example. That is, there must be a variable "ap" of type "va_list", and You must call "va_start" before calling **_frm_wri** as well as calling "va_end" before leaving the current context. Also note that the argument to "va_start" must always be the formal immediately to the left of the variable argument list (...).

* * *

Appendix C: Proliferation Chip Support.

The Archimedes C-51 software kit supports development of any chip based on the MCS-51 basic architecture, since all chips use the same basic instruction set. However, different chips have different special function registers.

The assembler has the basic 8051 chip's byte and bit registers pre-defined, whereas other family members must have the additional registers defined by EQU statements.

From C you access internal RAM and special function registers through special 8051 in-line functions (see section 1.17).

In header file **io51.h** the 8051/52/31/32 bit and byte register addresses are pre-defined. The header file **io515.h** include pre-defined addresses for the Siemens 80515/535 family. Other 8051 family members may require that you add #define statements for the additional registers featured.

Appendix D: C Error and Warning Messages

The error messages produced by the C-compiler falls into six categories:

1. Command line errors
2. Compilation warning messages
3. Compilation error messages
4. Compilation fatal error messages
5. Memory overflow message
6. Compiler internal errors

Command line errors

Command line errors occur when the compiler is invoked with bad parameters. The most common situation is that a file could not be opened. The command line interpreter is very strict about duplicate, misspelled or missing command line switches. However, it produces messages pointing out the problem in detail.

Compilation warning messages

Compilation warning messages are produced when the compiler has found a construct which typically is due to a programming error or omission. This appendix lists all warning messages. (Also, see section 1.20 regarding the usage of the -V switch).

Compilation error messages

Compilation error messages are produced when the compiler has found a construct which clearly violates the C language rules. Note that the Archimedes C-compiler is more strict on compatibility issues than many other C-compilers. In particular pointers and integers are considered as incompatible when not explicitly casted. This appendix lists all error messages. (Also, see section 1.20 regarding the usage of the -V switch.)

Compilation fatal errors

Compilation fatal error messages are produced when the compiler has found a user error so severe that further processing is not considered meaningful. After the diagnostic message has been issued the compilation is immediately terminated. This appendix lists all compilation error messages (some marked as fatal). (Also, see section 1.20 regarding the usage of the -V switch).

Memory overflow message

The Archimedes C-compiler is a memory-based compiler that in case of a system with a small primary memory or in case of very large source files may run out of memory. This is recognized by a special message:

```
***COMPILER OUT OF MEMORY***

      Dynamic memory used: nnnnnn bytes
```

If such a situation occurs the cure is either to add system memory or to split source files into smaller modules. However, with 512K RAM the compiler capacity is sufficient for all reasonably sized source files. If memory is on the limit please note that the **-q -x** and **-P** command line switches cause the compiler to use more memory

Compiler internal errors

During compilation a number of internal consistency checks are performed and if any of these checks fail the compiler will terminate after giving a short description of the problem. Such errors should normally not occur and should be reported to Archimedes technical support group.

The compiler returns status information to DOS which can be used in conjunction with IF ERRORLEVEL in .BAT files.

0 Compilation successful
1 There were warnings (returns 0 if -w switch is on)
2 There were errors
3 Fatal error detected

Error [0]: Invalid syntax
Compiler could not decode statement or declaration.

Error [1]: Too deep #include nesting (max is 10)
Fatal. Compiler limit for nesting of #include files exceeded. Recursive #include could be the reason.

Error [2]: Failed to open #include file 'name'
Fatal. Could not open #include file. File does not exist in specified directories (check -I prefixes and environment variable C_INCLUDE) or is disabled for reading.

Error [3]: Invalid #include file name
Fatal. #include file name must be written <file> or "file".

Error [4]: Unexpected end of file encountered
Fatal. End of file encountered within declaration, function definition or during macro expansion. Probable cause is bad () or {} nesting.

Error [5]: Too long source line (max is 512 characters); truncated
Source line length exceeds compiler limit.

Error [6]: Hexadecimal constant without digits
Prefix "0x" or "0X" of hexadecimal constant found without following hexadecimal digits.

Error [7]: Character constant larger than "long"
Character constant contains too many characters to fit in space of a long integer.

Error [8]: Invalid character encountered: '\xhh'; ignored
Character not included in 'C' character set was found.

Error [9]: Invalid floating point constant
Too large or invalid syntax of floating point constant. See ANSI standard for legal forms.

Error [10]: Invalid digits in octal constant
Non-octal digit in octal constant. Valid octal digits are: 0, 1, 2, 3, 4, 5, 6 and 7.

Error [11]: Missing delimiter in literal or character constant
No closing delimiter ' or " was found in character or literal constant.

Error [12]: String too long (max is 509)
Compiler limit for length of single or concatenated strings exceeded.

Error [13]: Argument to #define too long (max is 512)
Lines terminated by '\' resulted in too long #define line.

Error [14]: Too many formal parameters for #define (max is 127)
Fatal. Too many formal parameters in macro definition (#define directive).

Error [15]: ',' or ')' expected
Invalid syntax of function definition header or macro definition.

Error [16]: Identifier expected
Identifier is missing in declarator, "goto" statement or in pre-processor line.

Error [17]: Space or tab expected
Pre-processor arguments must be separated from the directive with tab or space characters.

Error [18]: Macro parameter 'name' redefined
A #defined symbol's formal parameter was repeated.

Error [19]: Unmatched #else, #endif or #elif
Fatal. Missing #if, #ifdef or #ifndef.

Error [20]: No such pre-processor command: 'name'
'#' was followed by an unknown identifier.

Error [21]: Unexpected token found in pre-processor line
Pre-processor line was not empty after the argument part was read.

Error [22]: Too many nested parameterized macros (max is 50)
Fatal. Pre-processor limit exceeded.

Error [23]: Too many active macro parameters (max is 256)
Fatal. Pre-processor limit exceeded.

Error [24]: Too deep macro nesting (max is 100)
Fatal. Pre-processor limit exceeded.

Error [25]: Macro 'name' called with too many parameters
Fatal. A parameterized #define macro was called with more arguments than declared.

Error [26]: Actual macro parameter too long (max is 512)
A single macro argument may not exceed the length of a source line.

Error [27]: Macro 'name' called with too few parameters
A parameterized #define macro was called with fewer arguments than declared.

Error [28]: Missing #endif
Fatal. End of file encountered during skipping of text after a false condition.

Error [29]: Type specifier expected
Type description missing. It could happen in struct, union, prototyped function definitions/declarations or in K&R function formal parameter declarations.

Error [30]: Identifier unexpected
Invalid identifier. It could be an identifier in a type name definition like:

```
sizeof ( int *ident );
```

or two consecutive identifiers.

Error [31]: Identifier 'name' redeclared
Redeclaration of declarator identifier.

Error [32]: Invalid declaration syntax
Undecodable declarator.

Error [33]: Unbalanced '(' or ')' in declarator
Paranthesis error in declarator.

Error [34]: Attribute(s) given for "void" type; ignored
Attribute "const" or "volatile" given with type specifier "void"; ignored.

Error [35]: Invalid declaration of "struct", "union" or "enum" type
"struct", "union" or "enum" was followed by invalid token(s).

Error [36]: Tag identifier 'name' redeclared
"struct", "union" or "enum" tag is already defined in the current scope.

Error [37]: Function 'name' declared within "struct" or "union"
Function declared as member of "struct" or "union".

Error [38]: Invalid width of field (max is nn)
Declared width of field exceeds the size of an integer.

Error [39]: ',' or ';' expected
Missing ',' or ';' at the end of declarator:

```
char c cc
int i;
```

Error [40]: Array dimension outside of "unsigned" "int" bounds
Array dimension negative or larger than can be represented in an unsigned integer.

Error [41]: Member 'name' of "struct" or "union" redeclared
Member of "struct" or "union" redeclared.

Error [42]: Empty "struct" or "union"
Declaration of "struct" or "union" containing no members.

Error [43]: Object cannot be initialized
Initialization of "typedef" declarator or "struct" or "union" member.

Error [44]: ';' expected
A statement or declaration needs a terminating semicolon.

Error [45]: ']' expected
Bad array declaration or array expression.

Error [46]: ':' expected
Missing colon after "default", "case" label or in '?'-operator.

Error [47]: '(' expected
Probable cause is a misformed "for", "if" or "while" statement.

Error [48]: ')' expected
Probable cause is a misformed "for", "if" or "while" statement or expression.

Error [49]: ',' expected
Invalid declaration.

Error [50]: '{' expected
Invalid declaration or initializer.

Error [51]: '}' expected
Invalid declaration or initializer.

Error [52]: Too many local variables and formal parameters (max is 1024)
Fatal. Compiler limit exceeded.

Error [53]: Declarator too complex (max is 128 '(' and/or '*')
Declarator contained too many '(', ')' or '*'.

Error [54]: Invalid storage class
Invalid storage-class for the object specified.

Error [55]: Too deep block nesting (max is 50)
Fatal. Too deep {} nesting in function definition.

Error [56]: Array of functions
Array of functions. The valid form is array of pointers to functions:

```
int array [ 5 ] ();        /* Invalid */

int (*array [ 5 ]) ();     /* Valid */
```

Error [57]: Missing array dimension specifier
Multi-dimensional array declarator without specified dimension. Only the first dimension can be excluded (in declarations of "extern" arrays and function formal parameters).

Error [58]: Identifier 'name' redefined
Redefinition of declarator identifier.

Error [59]: Function returning array
Function cannot return array.

Error [60]: Function definition expected
K&R function header found without following function definition:

```
int f ( i );
```

Error [61]: Missing identifier in declaration
Declarator lacks an identifier.

Error [62]: Simple variable or array of a "void" type
Only pointers, functions and formal parameters can be of "void" type.

Error [63]: Function returning function
Function cannot return function:

```
int f()();
```

Error [64]: Unknown size of variable object 'name'
Defined object has unknown size. It could be an external array with no dimension given or an object of an only partially (forward) declared "struct" or "union".

Error [65]: Too many errors encountered (>100).
Fatal. Compiler aborts after a certain number of diagnostic messages.

Error [66]: Function 'name' redefined
Multiple definitions of function encountered.

Error [67]: Tag 'name' undefined
Definition of variable of "enum" type with type undefined or reference to undefined "struct" or "union" type in function prototype or as "sizeof" argument.

Error [68]: "case" outside "switch"
"case" without any active "switch" statement.

Error [69]: Too many "case" labels (max is 512)
Too many "case" labels in a single "switch" statement.

Error [70]: Duplicated "case" label: nn
The same constant value used more than once as a "case" label.

Error [71]: "default" outside "switch"
"default" without any active "switch" statement.

Error [72]: Multiple "default" within "switch"
More than one "default" in one "switch" statement.

Error [73]: Missing "while" in "do" - "while" statement
Probable cause is missing {} around multiple statements.

Error [74]: Label 'name' redefined
Label defined more then once in the same function.

Error [75]: "continue" outside iteration statement
"continue" outside any active "while", "do - while" or "for" statement.

Error [76]: "break" outside "switch" or iteration statement
"break" outside any active "switch", "while", "do - while" or "for" statement.

Error [77]: Undefined label 'name'
There is "goto label" with no "label:" definition within function's body.

Error [78]: Pointer to a field not allowed
Pointer to a field member of "struct" or "union":

```
struct
  {
    int *f:6;
  }
```

Error [79]: Argument of binary operator missing
First or second argument of binary operator is missing.

Error [80]: Statement expected
One of '?', ':', ',', ']'or '}' in place where statement is expected.

Error [81]: Declaration after statement
Declaration was found after statement. Note that a single ';' is considered to be an empty statement:

```
int i;;     /* Second ';' is a STATEMENT so ...  */
char c;     /* this is a declaration after statement */
```

Error [82]: "else" without preceding "if"
Probable cause is bad {} nesting.

Error [83]: "enum" constant(s) outside "int" or "unsigned" "int" range
Too small or too large enumeration constant created.

Error [84]: Too deep "switch" nesting (max is 50)
Fatal. Too many nested "switch" statements.

Error [85]: Empty "struct", "union" or "enum"
Definition of "struct" or "union" that contains no members or definition of "enum" that contains no enumeration constants.

Error [86]: Invalid formal parameter

Declaration of K&R function with formal parameter(s) specified in the function header:

```
int f( c )              /* OK */
char c;
  {
    int g( cc );        /* Invalid */
```

Error [87]: Redeclared formal parameter: 'name'

A formal parameter in K&R function definition was declared more than once.

Error [88]: Contradictory function declaration

"void" appears in a function parameter type list together with other type of specifiers.

Error [89]: "..." without previous parameter(s)

"..." cannot be the only parameter description specified:

```
int f( ... );               /* Error */

int g( int, ... );          /* OK */
```

Error [90]: Formal parameter identifier missing

No identifier of parameter specified in the header of prototyped function definition:

```
int f( int *p, char, float ff )
          /* Second parameter has no name */
  {
    /* Function body */
  }
```

Error [91]: Redeclared number of formal parameters

Prototyped function declared with other number of parameters than the first time:

```
int f( int, char );
int f( int );               /* Less parameters */
int f( int, char, float );  /* More parameters */
```

Error [92]: Prototype appeared after reference

Prototyped declaration of a function after it was defined or referenced as K&R function.

Error [93]: Initializer to field of width nn (bits) out of range
Bit field initialized with constant too large to fit in field's space.

Error [94]: Fields of width 0 must not be named
Zero length fields are only used to align fields to the next "int" boundary and cannot be accessed via an identifier.

Error [95]: Too large difference between "case" values (max is 2000)
Difference between the smallest and the largest "case" value exceeds compiler limit.

Error [96]: "case" label out of range (>10000 or <-10000)
Value of "case" label exceeds compiler limit.

Error [97]: Undefined "static" function 'name'
Function was declared with "static" storage class but never defined.

Error [98]: Primary expression expected
Missing expression.

Error [99]: Tag identifier 'name' was never defined
Structure or union was forward declared but never defined.

Error [100]: Undeclared identifier: 'name'
Reference to identifier other than function that was not declared.

Error [101]: First argument of '.' operator must be of "struct" or "union" type
Dot operator '.' applied to argument that is not "struct" or "union".

Error [102]: First argument of '->' was not pointer to "struct" or "union"
Arrow operator '->' applied to argument that is not pointer to "struct" or "union".

Error [103]: Invalid argument of "sizeof" operator
"sizeof" operator applied to bit field, function or extern array of unknown size.

Error [104]: Initializer "string" exceeds array dimension
Array of "char" with explicit dimension given was initialized with a string exceeding array size:

```
char array [ 4 ] = "abcde";
```

Error [105]: Language feature not implemented: 'name'
The code-generator currently lacks the specified function.

Error [106]: Too many function parameters (max is 127)
Fatal. Too many parameters in function declaration/definition.

Error [107]: Function parameter 'name' already declared
Formal parameter in function definition header declared more than once:

```
int f( i, i )                   /* K&R function */
int i;
  {
  }

int f( int i, int i )           /* Prototyped function */
  {
  }
```

Error [108]: Function parameter 'name' declared but not found in header
Occurs in K&R function definition. Parameter declared but not specified in the function header:

```
int f( i )
int i, j     /* j is not specified in function header */
  {
  }
```

Error [109]: ';' unexpected
Unexpected delimiter.

Error [110]: ')' unexpected
Unexpected delimiter.

Error [111]: '{' unexpected
Unexpected delimiter.

Error [112]: ',' unexpected
Unexpected delimiter.

Error [113]: ':' unexpected
Unexpected delimiter.

Error [114]: '[' unexpected
Unexpected delimiter.

Error [115]: '(' unexpected
Unexpected delimiter.

Error [116]: Integral expression required
Bad type of expression - evaluated expression must have integral type.

Error [117]: Floating point expression required
Bad type of expression - evaluated expression must have floating type.

Error [118]: Scalar expression required
Bad type of expression - evaluated expression must have scalar type.

Error [119]: Pointer expression required
Bad type of expression - evaluated expression must have pointer type.

Error [120]: Arithmetic expression required
Bad type of expression - evaluated expression must have arithmetic type.

Error [121]: Lvalue required
Expression do not evaluate to a memory address.

Error [122]: Modifiable lvalue required
Expression do not designate a variable object or is "const".

Error [123]: Prototyped function argument number mismatch
Prototyped function called with other number of arguments than declared.

Error [124]: Unknown "struct" or "union" member: 'name'
Reference to nonexistent member of "struct" or "union".

Error [125]: Attempt to take address of field
It is not possible to apply '&' on bit-fields.

Error [126]: Attempt to take address of "register" variable
It is not possible to apply '&' on a object with "register" storage class.

Error [127]: Incompatible pointers
Incompatible pointers. There must be full compatibility of objects that pointers point to. It means that if pointers point (directly or indirectly) to prototyped functions, a compatibility test is performed not only on return values but includes as well compatibility test on number of parameters and their types so sometimes incompatibility can be hidden quite deeply:

```
        char (*(*p1)[8])(int);
        char (*(*p2)[8])(float);
/* p1 incompatible with p2; incompatible function parameters */
```

Compatibility test includes also checking of dimensions of arrays if they appear in description of objects pointed to:

```
int (*p1)[8];       /* p1 incompatible with p2; */
int (*p2)[9];       /* array dimensions differ */
```

Error [128]: Function argument incompatible with its prototype
Function argument does not match declaration.

Error [129]: Incompatible operands of binary operator
Type conflict in binary operator.

Error [130]: Incompatible operands of '=' operator
Type conflict in assignment.

Error [131]: Incompatible "return" expression
Expression is incompatible with the "return" value declaration.

Error [132]: Incompatible initializer
Initializer expression is incompatible with the object to be initialized.

Error [133]: Constant value required
Expression was not constant in "case" label, #if, #elif, bit-field declarator, array declarator or static initializer.

Error [134]: Unmatching "struct" or "union" arguments to '?' operator
The second and third argument of the '?'-operator are different.

Error [135]: " pointer + pointer" operation
It is an error to add two pointers.

Error [136]: Redeclaration error
Current declaration is inconsistent with earlier declarations of the same object.

Error [137]: Reference to member of undefined "struct" or "union"
Only pointers may be declared pointing to undefined "struct" or "union" declarators.

Error [138]: "- pointer" expression
Pointer expression preceded by '-' in other context than "pointer - pointer"

Error [139]: Too many "extern" symbols declared (max is 256).
Fatal. Compiler limit exceeded.

Error [140]: "void" pointer not allowed in this context
A pointer expression like indexing involved a void pointer (element size unknown).

Error [141]: #error 'any message'
Fatal. The pre-processor directive #error was found which notifies that somthing must be defined at command-line in order to compile this module.

Warning [0]: Macro 'name' redefined
A #defined symbol was redeclared with different argument or formal list.

Warning [1]: Macro formal parameter 'name' is never referenced
A #define formal parameter never appeared in the argument string.

Warning [2]: Macro 'name' is already #undef
An #undef is performed on a non-macro symbol.

Warning [3]: Macro 'name' called with empty parameter(s)
A parameterized #defined macro was called with a zero-length argument.

Warning [4]: Macro 'name' is called recursively; not expanded
A recursive macro makes pre-processor stop further expansion.

Warning [5]: Undefined symbol 'name' in #if or #elif; assumed zero
It is considered as bad programming practice to assume that non-macro symbols should be treated as zeros in #if and #elif expressions. Use either:

```
#ifdef symbol  or  #if defined (symbol)
```

Warning [6]: Unknown escape sequence ('\c'); assumed 'c'
A backslash (\) found in a character constant or string literal was followed by an unknown escape character.

Warning [7]: Nested comment found without using the '-C' option
The character sequence /* was found within a comment. Ignored.

Warning [8]: Invalid type-specifier for field; assumed "int"
Bit-fields may in this implementation only be specified as "int" or "unsigned" "int".

Warning [9]: Undeclared function parameter 'name'; assumed "int"

An undeclared identifier in the header of a K&R function definition is by default set to "int".

Warning [10]: Dimension of array ignored; array assumed pointer

An array with an explicit dimension was specified as a formal parameter. It was rewritten to read: pointer to object.

Warning [11]: Storage class "static" ignored; 'name' declared "extern"

An object or function was first declared as "extern" (explicitly or by default) and later declared as "static" which is ignored.

Warning [12]: Incompletely bracketed initializer

Initializers should either be 'flat' (only one level of {}) or completely surrounded by curly brackets in order to avoid ambiguity.

Warning [13]: Unreferenced label 'name'

Label was defined but never referenced.

Warning [14]: Type specifier missing; assumed "int"

No type specifier given in declaration - assumed to be "int".

Warning [15]: Wrong usage of string operator ('#' or '##'); ignored

The current implementation restricts usage of '#' and '##' operators to the token-field of parameterized macros. In addition the '#' operator must precede a formal parameter.

```
#define mac(p1)          #p1          /* Becomes "p1" */
#define mac(p1,p2)      p1+p2##add_this
          /* Merged p2 */
```

Warning [16]: Non-void function: "return" with <expression>; expected

A non-void function definition is supposed to exit with a defined return value in all places.

Warning [17]: Invalid storage class for function; assumed to be "extern"

Invalid storage class for function - ignored. Valid is "extern", "static" or "typedef".

Warning [18]: Redeclared parameter's storage class

Storage class of a function formal parameter changed from "register" to "auto" or vice versa in a subsequent declaration/defintion.

Warning [19]: Storage class "extern" ignored; 'name' was first declared as "static"

An identifier declared as "static" was later explicitly or implicitly declared as "extern".

Warning [20]: Unreachable statement(s)

Statement(s) was preceded by an unconditional jump or return:

```
break;
i = 2;                  /* Never executed */
```

Warning [21]: Unreachable statement(s) at unreferenced label 'name'

Labeled statement(s) was preceded by an unconditional jump or return but the label was never referenced:

```
  break;
here:
   i = 2;                /* Never executed */
```

Warning [22]: Non-void function: explicit "return" <expression>; expected

A non-void function generated an implicit return. Could be the result of an unexpected exit from a loop or switch. Note that a "switch" without "default" is always supposed to be 'exitable' regardless of any "case" constructs.

Warning [23]: Undeclared function 'name'; assumed "extern" "int"

Reference to undeclared function causes default declaration. Function is assumed to be of K&R type, having "extern" storage class and returning "int".

Warning [24]: Static memory option converts local "auto" or "register" to "static"
A command line option for static memory allocation was activated which makes "auto" and "register" declarations read as "static".

Warning [25]: Inconsistent use of K&R function - varying number of parameters
K&R function called with changing number of parameters.

Warning [26]: Inconsistent use of K&R function - changing type of parameter
K&R function called with changing types of parameters:

```
f( 34 );        /* Integral argument */
f( 34.6 );      /* Floating argument */
```

Warning [27]: Size of "extern" object 'name' is unknown
The global type-check option requires that "extern" arrays are declared with size.

Warning [28]: Constant [index] outside array bounds
Constant index outside declared array bounds (< 0 or >=dim).

Warning [29]: Aggregate cannot be "register"
Objects of array, structure or union type cannot have "register" storage class.

Warning [30]: Attribute ignored
Since "const" or "volatile" are attributes of objects they are ignored when given with structure, union or enumeration tag definition with no objects declared at the same time:

```
const struct s
  {
    ...
  };        /* No object declared; "const" ignored */
```

Warning [31]: Incompatible parameters of K&R functions

Pointers (could be indirect) to functions or K&R function declarators directly used in one of following contexts:

"pointer - pointer",
"expression ? ptr : ptr",
"pointer relational_op pointer"
"pointer equality_op pointer",
"pointer = pointer" or
"formal parameter vs actual parameter"

have incompatible parameter types.

Warning [32]: Incompatible numbers of parameters of K&R functions

Pointers (could be indirect) to functions or K&R function declarators directly used in one of following contexts:

"pointer - pointer",
"expression ? ptr : ptr",
"pointer relational_op pointer"
"pointer equality_op pointer",
"pointer = pointer" or
"formal parameter vs actual parameter"

have different number of parameters.

Warning [33]: Local or formal 'name' was never referenced

Formal parameter or local variable object is unused in function definition.

Warning [34]: Non-printable character '\xhh' found in literal or character constant

It is considered as bad programming practice to use non-printable characters in string literals or character constants. Use \0xhh to get the same function.

Warning [35]: Old-style (K&R) type of function declarator

During compilation with the -gA option on, obsolete function declarators was found.

Appendix E: Assembler Error Messages

The list below shows the 8051 Assembler error messages.

Error: INTERNAL ALIGN ERROR
Should never occur, if no other error messages have been issued.

Error: Invalid character
Unrecognized token.

Error: Too long line
There must be no more than 132 characters in a line. This applies also to macro generated lines.

Error: Invalid digit
Bad numeric constant.

Error: ";" or end-of-line expected
Bad delimiter after the last operand or instruction.

Error: Value out of range
Address or data expression to large or to small.

Error: Syntax error
Undecodable statement.

Error: Expression too complex
Too deep stack needed to evaluate the expression. Re-arrange or simplify expression.

Error: Invalid label
See section "Label syntax".

Error: Duplicate label
Double definition.

Error: Undefined label: xxxxx
No definition.

Error: Invalid string constant
Bad characters in string constant. See general section "Integer, ASCII and real constants".

Error: Invalid instruction
No such instruction.

Error: Invalid operand combination
Valid operands, but not for this instruction.

Error: Missing END-statement
There must be and END somewhere.

Error: Type conflict
Something about the type (relocation mode) is wrong.

Error: Unmatched NAME, MODULE, END or ENDMOD
Like NAME followed by NAME, instead of END or ENDMOD.

Error: Unmatched MACRO or ENDMAC
Valid sequence is MACRO - ENDMAC.

Error: Bad ELSE or ENDIF nesting
See general section "Conditional assembly".

Error: Over or underflow in macro stack, v=nnn
Macro stack pointer is less than zero or greater than 100.

Error: Macro parameter out of range
Trying to access parameters outside P0-P9.

Error: Unknown "\" directive in macro
See general section "Macro processing".

Error: Multiple or invalid declaration
See general sections "Modules" and "Segments".

Error: Too deep file nesting, > 3
The limit on include files.

Error: No such file
Could not open include file. Stops assembler.

Error: Expression <> current relocation
Operand must belong to current segment.

Error: Expression is not absolute
Operand must be an absolute expression.

Error: Operator not allowed here
Bad use of SFx or LOW, HIGH, LWRD or HWRD operators. See general section "Expression and operators".

Error: ACALL or AJMP in relocatable section
Only allowed in ASEG.

Error: Too many externals or segments defined
256 is the limit.

Error: Stack overflow, error code 2000 PC=xxxx
Outo of symbol table space for this assembly module. Break module into more than one module.

Appendix F: Linker Error Messages.

There are several kind of linker error and warning messages. Most of these are listed below. In addition, you might get a memory overflow message. The Archimedes linker is a memory-based linker. As such, in case of a system with a small primary memory or very large source files, it may run out of memory. This is recognized by the special message:

```
***LINKER OUT OF MEMORY***
    Dynamic memory used: nnnnnn bytes
```

If such a situation occurs the cure is either to add system memory or to use the option '-m' (enable file bound processing, see *3.12 File bound processing*). However, with 512K RAM the linker capacity is sufficient for most reasonably sized object files.

During linking a number of internal consistency checks are performed and if any of these checks fail the linker will terminate after giving a short description of the problem. Such 'internal errors' should normally not occur and should be reported to Archimedes technical support group. Please include all possible information about the problem and preferably also a diskette with the program that generated the internal error.

Below most linker error and warning messages are listed.

Error [0]: Format chosen cannot support banking
Fatal. Format unable to support banking. See *3.13 Banking*.

Error [1]: Corrupt file. Unexpected end of file in module *module* **(** *file* **), encountered**
Fatal. Linker aborts immediately.

Error [2]: Too many errors encountered (>100)
Fatal. Linker aborts immediately.

Error [3]: Corrupt file. Checksum failed in module *module* **(** *file* **). Linker checksum is** *linkcheck*, **module checksum is** *modcheck*
Fatal. Linker aborts immediately.

Error [4]: Corrupt file. Zero length identifier encountered in module *module* (*file*)
Fatal. Linker aborts immediately.

Error [5]: Address type for cpu incorrect. Fault encountered in module *module* (*file*)
Fatal. Linker aborts immediately.

Error [6]: Program module *module* **declared twice, redeclaration in file** *file*. **Ignoring second module**
Not fatal. The linker will *not* produce code, unless option **-B** (forced dump) is used.

Error [7]: Corrupt file. Unexpected UBROF-format end of file encountered in module *module* (*file*)
Fatal. Linker aborts immediately.

Error [8]: Corrupt file. Unknown or misplaced tag encountered in module *module* (*file*). **Tag** *tag*
Fatal. Linker aborts immediately.

Error [9]: Corrupt file. Module *module* **start unexpected in file** *file*
Fatal. Linker aborts immediately.

Error [10]: Corrupt file. Segment no. *sno no* **declared twice in module** *module* (*file*)
Fatal. Linker aborts immediately.

Error [11]: Corrupt file. External no. *ext no* **declared twice in module** *module* (*file*)
Fatal. Linker aborts immediately.

Error [12]: Unable to open file *file*
Fatal. Linker aborts immediately.

Error [13]: Corrupt file. Error tag encountered in module *module* (*file*)
Fatal. A UBROF error tag was encountered. Linker aborts immediately.

Error [14]: Corrupt file. Local *local* defined twice in module *module* (*file*)
Fatal. Linker aborts immediately.

Error [15]: Faulty bank definition -b*bank def*
Fatal. Incorrect syntax. Linker aborts immediately.

Error [16]: Segment *segment* is too long for segment definition
Fatal. The segment defined does not fit in the memory area reserved for it. Linker aborts immediately.

Error [17]: Segment *segment* is defined twice in segment definition -Z*segdef*
Fatal. Linker aborts immediately.

Error [18]: Range error in module *module* (*file*), segment *segment* at address *address*. Value *value*, in tag *tag*, is out of bounds
Not fatal. The address is outside the cpu address range. How this can occur is shown in the following example:

A relative segment:

```
0000 nop
0001 nop
 .    .
 .    .
0025 nop
0026 JUMP LABEL1
0027 nop
 .    .

0FC6 LABEL1:
0FC7 nop
 .    .
```

If this segment is defined, or allocated, to start at address F700 then the address to LABEL1 will be F700 + 0FC7 = 106C7. If the cpu address range is 16-bits (range FFFF), it will be unable to reach LABEL1.
The segment will therefore partly be outside the cpu address range

This check can be suppressed by use of the switch -R (disable range check).

Error [19]: Corrupt file. Undefined segment referenced in module *module* **(** *file* **)**
Fatal. Linker aborts immediately.

Error [20]: Undefined external referenced in module *module* **(** *file* **)**
Fatal. Linker aborts immediately.

Error [21]: Segment *segment* **in module** *module* **does not fit bank**
Fatal. The segment is to long. Linker aborts immediately.

Error [22]: Paragraph no. is not applicable for the wanted cpu. Tag encountered in module *module* **(** *file* **)**
Fatal. Linker aborts immediately.

Error [23]: Corrupt file. T_REL_FI_8 or T_EXT_FI_8 is corrupt in module *module* **(** *file* **)**
Fatal. The tag *T_REL_FI_8* or *T_EXT_FI_8* is faulty. Linker aborts immediately.

Error [24]: Segment *segment* **overlap segment** *segment*
The segments overlap each other. i.e both have code on the same address.

Error [25]: Corrupt file. Unable to find module *module* **(** *file* **)**
Fatal. A module is missing. Linker aborts immediately.

Error [26]: Segment *segment* **is to long**
Fatal. This error should never occur, unless the program is huge. Linker aborts immediately.

Error [27]: Several entries: *entry* **exists**
Fatal. There are two or more entries with the same name. Linker aborts immediately.

Error [28]: File *file* **is too long**
Fatal. This error should never occur unless the program is huge. Linker aborts immediately.

Error [29]: No object file specified in command-line
Fatal. There is nothing to link. Linker aborts immediately.

Error [30]: Option *-op* also requires the *-op* option
Fatal. Linker aborts immediately.

Error [31]: Option *-op* cannot be combined with the *-op* option
Fatal. Linker aborts immediately.

Error [32]: Option *-op* cannot be combined with the *-op* option and the *-op* option
Fatal. Linker aborts immediately.

Error [33]: Faulty value *val in command line or in XLINK_PAGE*, (range is 10-150)
Fatal. Faulty page setting. Linker aborts immediately.

Error [34]: Filename is too long
Fatal. The filename is more than 255 characters long. Linker aborts immediately.

Error [35]: Unknown flag *flag* in xref option *xref op*
Fatal. Linker aborts immediately.

Error [36]: Option *op* does not exist
Fatal. Linker aborts immediately.

Error [37]: - not succeed by character
Fatal. The '-' marks the beginning of an option, and must be followed by a character. Linker aborts immediately.

Error [38]: Option *op* is defined several times
Fatal. Linker aborts immediately.

Error [39]: Illegal character specified in option *op*
Fatal. Linker aborts immediately.

Error [40]: Argument expected after option *op*
Fatal. This option must be succeeded by an argument. Linker aborts immediately.

Error [41]: Unexpected '-' in option *op*
Fatal. Linker aborts immediately.

Error [42]: Faulty symbol definition -D*symb def*
Fatal. Incorrect syntax. Linker aborts immediately.

Error [43]: Symbol in symbol definition is too long
Fatal. The symbol name is more than 255 characters. Linker aborts immediately.

Error [44]: Faulty value *val in command line or in XLINK_COLUMN*, **(range 80-300)**
Fatal. Faulty column setting. Linker aborts immediately.

Error [45]: Unknown cpu *cpu* **encountered in** *command line or in XLINK_CPU*
Fatal. Linker aborts immediately.

Error [46]: Undefined external *external* **referred in** *module* (*file*)
Not fatal. Entry to this external is missing.

Error [47]: Unknown format *format* **encountered in** *command line or XLINK_FORMAT*
Fatal. Linker aborts immediately.

Error [48]: Faulty segment definition -Z*segdef*
Fatal. Incorrect syntax. Linker aborts immediately.

Error [49]: Segment name in segment definition is too long
Fatal. The segment name is more than 255 characters long. Linker aborts immediately.

Error [50]: Paragraph no. not allowed for this cpu, encountered in option *op*
Fatal. Linker aborts immediately.

Error [51]: *Hexadecimal or decimal* **value expected in option** *op*
Fatal. Linker aborts immediately.

Error [52]: Overflow on value in option *op*
Fatal. Linker aborts immediately.

Error [53]: Parameter exceeded 255 characters in extended command line file *file*
Fatal. Linker aborts immediately.

Error [54]: Extended command line file *file* **is empty**
Fatal. Linker aborts immediately.

Error [55]: Extended command line variable XLINK_ENVPAR is empty
Fatal. Linker aborts immediately.

Error [56]: Overlapping ranges in segment definition *segment def*
Fatal. Linker aborts immediately.

Error [57]: No cpu defined
Fatal. No cpu defined, neither in command line nor in XLINK_CPU. Linker aborts immediately.

Error [58]: No format defined
Fatal. No format defined, neither in command line nor in XLINK_FORMAT. Linker aborts immediately.

Error [59]: Revision no. for file is incompatible with XLINK revision no.
Fatal. Linker aborts immediately.

Error [60]: Segment *segment* **defined in bank definition and segment definition.**
Fatal. Linker aborts immediately.

Error [61]: Symbol in bank definition is too long
Fatal. Linker aborts immediately.

Error [62]: File *file* **multiple defined in command line**
Fatal. Linker aborts immediately.

Corrupt file means that the file to be linked is, in some way, faulty. The usual solution is simply to re-compile the file and try again, the fault that occurred may have been a simple read error, a transmission error, linking with old, faulty files etc.

Warning [0]: Too many warnings
Too many warnings encountered.

Warning [1]: Error tag encountered in module *module* **(** *file* **)**
An *UBROF* error tag is encountered when loading file *file*. This indicates a corrupt file and will generate an error in the linking phase.

Warning [2]: Symbol *symbol* **is redefined in command-line**
The linker warns on redefinition of symbols.

Warning [3]: Type conflict. Segment *segment*, **in module** *module*, **is incompatible with earlier segment(s) of the same name**
Segment of the same name *should* have the same type.

Warning [4]: Close/open conflict. Segment *segment*, **in module** *module*, **is incompatible with earlier segment of the same name**
Segments of the same name *should be either* open *or* closed.

Warning [5]: Segment *segment* **cannot be combined with previous segment**
The segments will not be *combined.*

Warning [6]: Type conflict for external/entry *entry*, **in module** *module* **against external/entry in module** *module*
Entries and their corresponding externals *should* have the same type.

Warning [7]: Module *module* **declared twice, once as program, and once as library. Redeclared in file** *file*, **ignoring library module**
The program module is linked.

Warning [8]: Segment *segment* **undefined in segment or bank definition**
Undefined segment exists. All segments *should* be defined in either the segment or the bank definition

Warning [9]: Ignoring redeclared program entry
Only the first found program entry is chosen.

Warning [10]: No modules to link
The linker has no modules to link.

Warning [11]: Module *module* declared twice as library. Redeclared in file *file*, ignoring second module
The first found module is linked.

Warning [12]: Using SFB in banked segment *segment* in module *module* (*file*)
The SFB assembler directive may not work in a banked segment.

Warning [13]: Using SFE in banked segment *segment* in module *module* (*file*)
The SFE assembler directive may not work in a banked segment.

Appendix G: Librarian Error Messages.

Error messages are of two kinds, command errors and fatal errors. Command errors only abort the current command, while fatal errors make XLIB abort. In addition to the messages below, Pascal (the implementation language) I/O errors may also occur. These messages are normally easy to interpret. Commands flagged as erroneous never alter object files!

Error: Bad object file, EOF encountered
Fatal. Bad or empty object file, could be the result of an aborted assembly.

Error: Unexpected EOF in batch file
Fatal. The last command in a command file must be EXIT.

Error: File open error
Fatal. Could not open the command file, or if ON-ERROR-EXIT has been specified, this message is issued on any failing file open operation.

Error: Variable length record out of bounds
Fatal. Bad object module, could be the result of an aborted assembly.

Error: Missing or non-default parameter
A parameter was missing in the "direct mode". Look into section 4.11

Error: No such CPU
A list with the possible choices is listed when this error is found. 8051 family CPU must be specified.

Error: CPU undefined
DEFINE-CPU must be issued before object file operations can begin. A list with the possible choices is listed when this error is found. 8051 family CPU must be specified.

Error: Ambiguous CPU type
A list with the possible choices is listed when this error is found. 8051 family CPU must be specified.

Error: No such command
Use the HELP command or look into section 4.11

Error: Ambiguous command
Use the HELP command or look into section 4.11

Error: Invalid parameter(s)
Too many parameters or a misspelled parameter.

Error: Module out of sequence
Fatal. Bad object module, could be the result of an aborted assembly.

Error: Incompatible object.
Fatal. Bad object module, could be the result of an aborted assembly, or that the used assembler/compiler revision, is incompatible with the version of XLIB used.

Error: Unknown tag: hh
Fatal. Bad object module, could be the result of an aborted assembly.

Error: Too many errors
Fatal. More than 32 errors will make XLIB abort.

Error: Assembly/compilation error?
Fatal. The T_ERROR tag was found. Edit and re-assemble/compile your program.

Error: Bad CRC, hhhh expected
Fatal. Bad object module, could be the result of an aborted assembly.

Error: No help file!
Fatal. Could not find the file with the help text. Look into section 4.13 for the fix.

Error: Can't find module: xxxxx
Check what's available with LIST-MOD THEFILE.

Error: Module expression out of range
Module expression is less than one or greater than <$>.

Error: Bad syntax in module expression: xxxxx
Look into section 4.6

Error: No SCRATCH!
Fatal. Could not create a temporary file. Possible cause: Directory is full.

Error: INTERNAL ERROR IN PROCEDURE: xxx/nnn,
Fatal. If no other fatal errors have occurred so far, this should never happen.

Error: <End module> found before <Start module>!
Source module range must be from low to high order.

Error: Before or after!
Bad <Before/After> specifier in the INSERT-MODULES command.

Error: Illegal insert sequence
The specified <Dest mod> in the INSERT-MODULES command, must not be within the <Start mod> - <End mod> sequence.

Appendix H: Other Development Tools Support.

Please review the colored release notes for the most up-to-date information.

Appendix I: Code Example

This Appendix contains listings of an elaborate code example. The code example is discussed in details in the TUTORIAL section.

EXAMPLE.C

```
/* EXAMPLE.C:  Some sample code for the Archimedes C-51 compiler kit.
 * See the file EXAMPLE.DOC and the Tutorial Introduction to the
 * manual for more information.  Revised 9-14-87, VC.  */

#include <stdio.h>      /* required header for printf, putchar */
#include <io51.h>       /* required header for output, bit_set, etc. */

#define TRUE 1
#define FALSE 0
#define BELL 7          /* ASCII terminal bell */

/* The following is a function prototype for the assembly
 * language routine, pulse.  See the source file PULSE.S03 */

extern int pulse(int count, char value);


/* ------------------------------------------------------- */
void set_timer()   /* initialize Timer 0 to generate interupts */
{
  output(TMOD,1);       /* Timer-0: mode 1 (16-bit timer) */
  output(TH0,0x4C);     /* load 4C00h for interrupt every 50 ms */
  output(TL0,0);
  set_bit(TR0_bit);             /* set Timer 0 run control bit */
  set_bit(ET0_bit);             /* set interrupt mask bit ET0 */
  set_bit(EA_bit);              /* enable interrupts bit EA */
}
```

```
/* ------------------------------------------------------- */
void ring_bell()   /* Timer-0 interrupt handler, rings the CRT
                    * bell approximately every 3 seconds; called
                    * from vector in CSTARTUP module */
{
  static int intr_ctr = 0;  /* counter for number of interrupts */
  output(TH0,0x4C);     /* reload timer again, hi byte */
  output(TL0,0);        /* low byte */
  if(++intr_ctr == 60)  /* only do every 60 interrupts */
  {
    putchar(BELL);      /* ring the bell, or do some useful task */
    intr_ctr = 0;
  }
}


/* ------------------------------------------------------- */

#define BIT_VAR 0       /* 1st bit address in internal RAM (20.0 hex) */

void do_io()     /* demonstrate 8051 I/O functions */
{
  char c;

    /* write and read Port 1 */
  output(P1,0x0F);
  printf("\nPort 1: %02X\n",input(P1));
        /* Write and read an absolute external data address,
     * such as a UART or other memory-mapped I/O device.
     * "(char *)" casts the int constant to a char pointer. */
  *(char *)0xE000 = c;   /* write */
  c = *(char *)0xE000;   /* read */
  c = read_XDATA(0xE00); /* another way, for SMALL model */
  printf("Data at E000h = %02Xh\n",c);
    /* read a byte of CODE memory */
  printf("Location 0 in code memory is: %02X",read_CODE(0));
        /* show use of bit-addressable RAM variables */
  if(read_bit(P1_0_bit) || read_bit(P1_1_bit))
    set_bit(BIT_VAR);
}
```

```
/* ---------------------------------------------------------
 * Eratosthenese Sieve program from BYTE, 1/83  */

#define SIZE 8190                /* size of array for sieve routine */
char flags[SIZE+1];      /* array for sieve */

void sieve()
{
  register int i,k;      /* register class NOT supported -- ignored */
  int prime,count,iter;

  printf("\nSieve: 10 iterations...\n");
  for (iter = 1; iter <= 10; iter++)        /* do program 10 times */
  {
    count = 0;           /* initialize prime counter */
    for (i = 0; i <= SIZE; i++)   /* set all flags true */
      flags[i] = TRUE;
    for (i = 0; i <= SIZE; i++)
    {
      if (flags[i])      /* found a prime */
      {
        prime = i + i + 3;         /* twice index + 3 */
        for (k = i + prime; k <= SIZE; k += prime)
          flags[k] = FALSE;        /* kill all multiples */
        count++;         /* primes found */
        }
      }
  }
  printf("%d primes, sieve done.\n",count); /* found in 10th pass */
}
/* ---------------------------------------------------------- */
void main()   /* Main EXAMPLE.C program */
{
  printf("EXAMPLE.C sample program.\n");
  set_timer();           /* turn on timer and interrupts */
  printf("Timer enabled, bell should ring every 3 seconds.\n\n");
  do_io();                /* display RAM, Port1, etc. */
  printf("\nPulsing Port 1.7 (8051 pin 8)\n");
  pulse(200,0xA5);       /* call assembler routine pulse P1 */
  sieve();               /* do the Sieve benchmark */
}
```

PULSE.S03

```
;------------------------------------------------------
;
;       PULSE.S03
;       int pulse(int count, byte value);
;
; This is a sample routine called by EXAMPLE.C to
; demonstrate how to call assembly routines from C.
;
; This function is invoked with an integer (count)
; loaded in R2:R3 as the first formal C parameter
; and a byte (value) as the second parameter, which
; is on the top of the C stack.
; The 8051 I/O pin P1.7 is pulsed for count number of
; cycles at maximum frequency, then value is written
; to Port 1.  It returns the current value of Port 3.
;
;------------------------------------------------------

        MODULE  pulse
        PUBLIC  pulse
        EXTERN  ?POP_R3_L17
        RSEG    CODE
pulse:
                                ; count is in R2/R3
        CLR     P1.7            ; turn off/on P1.7
        NOP
        SETB    P1.7
        DJNZ    R3,pulse; using only lo-byte for count
        LCALL   ?POP_R3_L17     ; get second parameter into R5
        MOV     P0,R3           ;  write to P1
        MOV     R3,P3           ; return with int value of
        MOV     R2,#0           ;  Port 3 in R2-R3 pair
        RET

        END
```

EXAMPLE.XCL

```
-! EXAMPLE.XCL: XLINK command file for example program
                compiled with the large (-m0) memory model. -!

-!   Define the CPU type... -!
-c8051
-!   Assign/allocate a value for the internal stack pointer... -!
-Z(IDATA)ISTACK=7
-!   Select Register Bank 0... -!
-D_R=0
-!   Set the code and read-only data segments to start at the
     beginning of ROM, usually at zero... -!
-Z(CODE)CSTART,RCODE,CODE,CDATA,ZVECT,CONST,CSTR,CCSTR=0
-!   Set the writable data segments to the beginning of the
     external RAM area (zero, in this case).  In the "large"
     memory model this area may overlap the read-only segments
     since they denote a different physical memory space
     (code vs. data)... -!
-Z(XDATA)DATA,TEMP,IDATA,UDATA,ECSTR,WCSTR,XSTACK=0
-!   Load the main C program, pulse assembler routine, C-library... -!
example
pulse
cl8051
-!    Generate a cross-reference/map file to EXAMPLE.MAP... -!
-x
-l example.map
-!    Disable the overlapping address check (because of the 8051's
     seprate code and data spaces... -!
-z
-!    Create output file EXAMPLE.A03 in Intel object module format
     for the ICE-51/5100 and compatible emulator... -!
-FAOMF8051
-o example.a03
```

CSTARTEX.S03

```
;------------------------------------------------;
;                                                ;
;                  EXAMPLE.S03                   ;
;                                                ;
; This module contains a modified version of the C ;
; startup routine adapted for the example described ;
; in file "example.doc"                          ;
;------------------------------------------------;
        NAME    CSTARTUP
        PUBLIC  exit
        EXTERN  main
        EXTERN  ring_bell       ; C interrupt handler
        EXTERN  _R              ; Register Bank

;------------------------------------------------------------;
; Virtual stack segment. It should be mapped into external RAM ;
;------------------------------------------------------------;
        RSEG    XSTACK
        DS      512             ; Change if needed
xstack_end:

;------------------------------------------------------------;
; Internal stack used for LCALL's and temporary storage for    ;
; code generator help-routines (math etc).  Stack can be loca- ;
; ted anywhere in the internal RAM with the exception of       ;
; _R - _R+7 that are reserved for R0-R7. Note that C interrupt ;
; routines require that you increase stack by 25 bytes to      ;
; assure that no overflows occur.                              ;
;------------------------------------------------------------;
        RSEG    ISTACK
stack_begin:
        DS      45              ; 25 bytes added to default

        RSEG    CSTART
startup:                ; Should be at location zero
        SJMP        init_C
        ORG         startup + 0Bh
        SJMP        handler
        ORG         startup + 30h
```

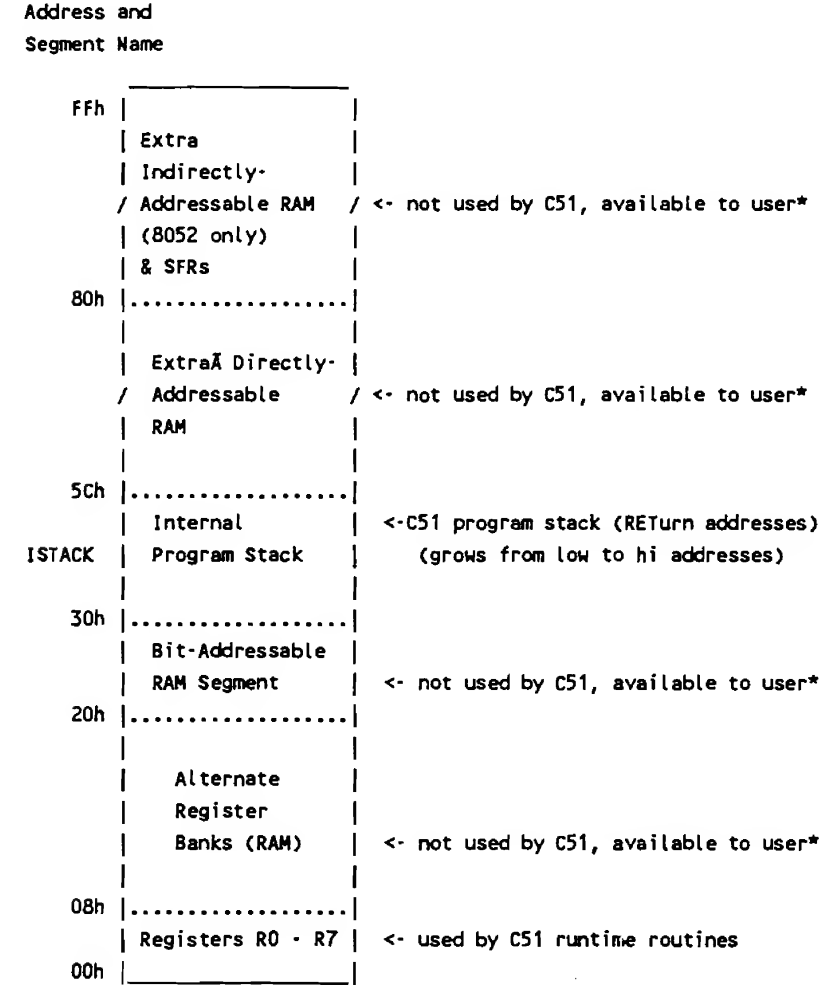
```
;------------------------------------------------------------;
; This is the place to insert interrupt vectors/handlers in.  ;
; For locations consult the Intel Microcontroller Handbook.   ;
;------------------------------------------------------------;

init_C:
;------------------------------------------------------------;
; Ativate the (at link-time) selected register bank.          ;
;------------------------------------------------------------;

        MOV     A,#_R
        MOV     C,ACC.3
        MOV     PSW.3,C
        MOV     C,ACC.4
        MOV     PSW.4,C
        MOV     SP,#stack_begin  ; From low to high addresses

;------------------------------------------------------------;
; If you don't want global/static C variables to be initial- ;
; ized at startup you can just remove the next two lines.     ;
;------------------------------------------------------------;

        EXTERN  ?SEG_INIT_L17
        LCALL   ?SEG_INIT_L17    ; Initialize segments

;------------------------------------------------------------;
; If hardware must be initiated from assembly or if interrupts ;
; should be on when reaching main, this is the place to insert ;
; such code.                                                  ;
;------------------------------------------------------------;

        MOV     R6,#HIGH(xstack_end)
        MOV     R7,#LOW(xstack_end)
        MOV     DPTR,#0          ; No parameters
        LCALL   main             ; main()
exit:                            ; someone called exit()
```

```
;-------------------------------------------------------------;
; Now when we are ready with our C program (usually 8051 C     ;
; programs are continouous) we must perform a system-dependent ;
; action.  In this simple case we just stop.                   ;
;-------------------------------------------------------------;

        SJMP    $               ; Forever...

handler:                        ; saves all regs., calls C routine
        PUSH    ACC
        PUSH    B
        PUSH    PSW
        PUSH    DPH
        PUSH    DPL
        PUSH    _R+0
        PUSH    _R+1
        PUSH    _R+2
        PUSH    _R+3
        PUSH    _R+4
        PUSH    _R+5
        MOV     DPTR,#0         ; 0 parameters pushed
        LCALL   ring_bell       ; C interrupt handler
        POP     _R+5
        POP     _R+4
        POP     _R+3
        POP     _R+2
        POP     _R+1
        POP     _R+0
        POP     DPL
        POP     DPH
        POP     PSW
        POP     B
        POP     ACC
        RETI

        END     startup
```

Appendix J: Memory Maps

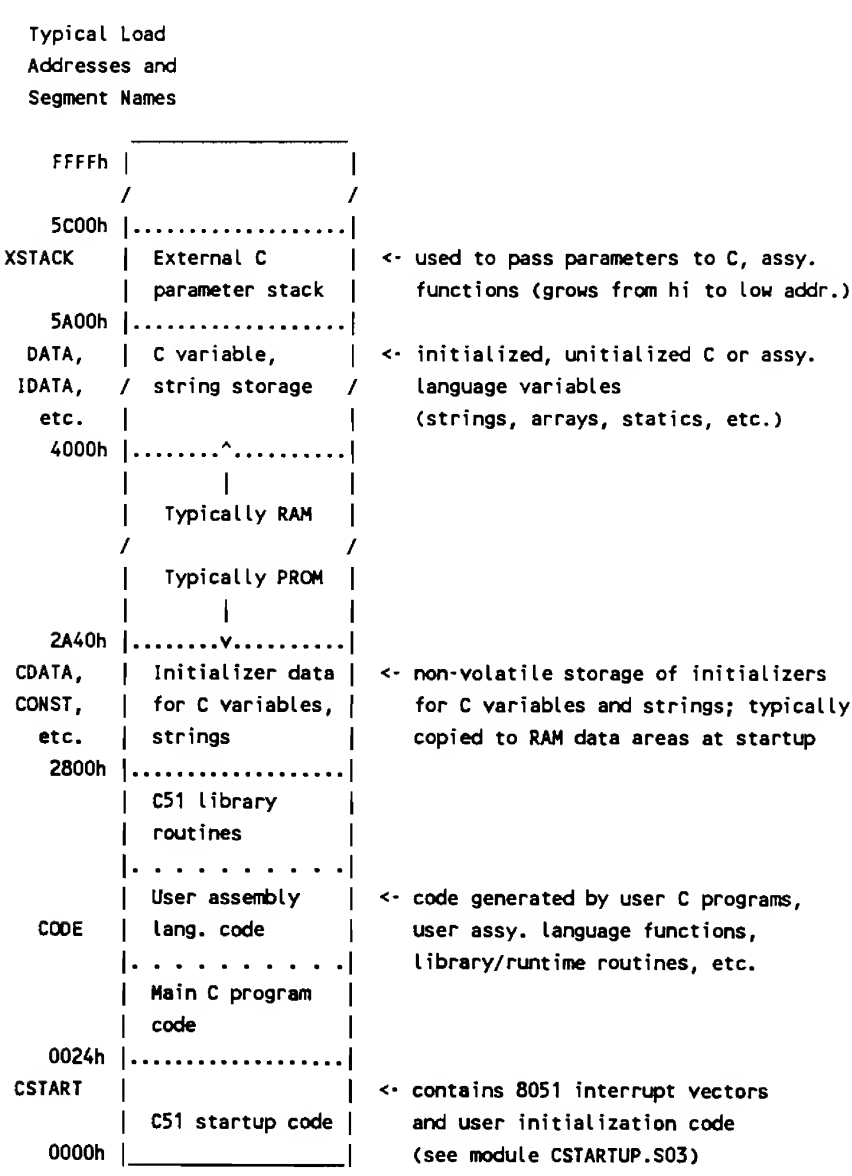
8031/8032 INTERNAL MEMORY ORGANIZATION WITH C-51. For Medium and Large Models.

This memory map assumes a medium or large model program with the stack starting at location 30h (SP initialized to 2Fh), and a stack segment size of 45 (ISTACK). Stack location and size can be easily altered.



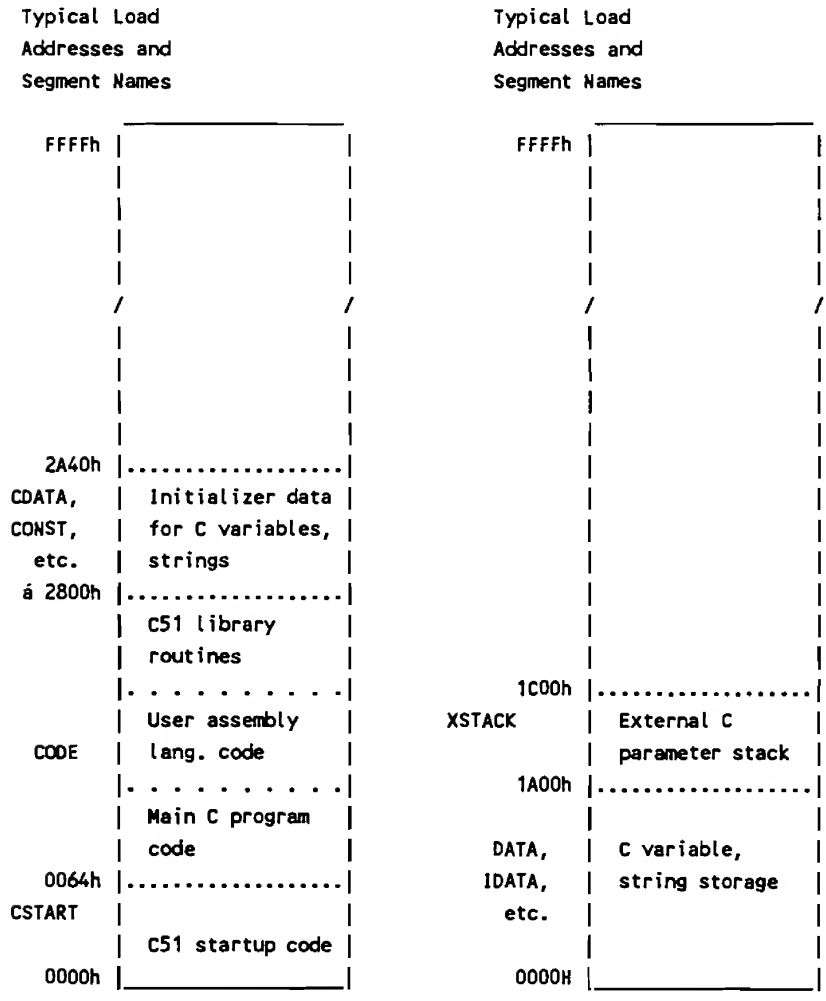
8031/8032 EXTERNAL MEMORY ORGANIZATION WITH C-51. For Medium Memory Models. (64K CODE/DATA, /PSEN ANDed to /RD)

Combined CODE (program) and DATA Memory Map



8031/8032 EXTERNAL MEMORY ORGANIZATIOH WITH C-51. For Large Memory Models (64K CODE + 64K DATA).

CODE and DATA segments both linked at starting address 0000.

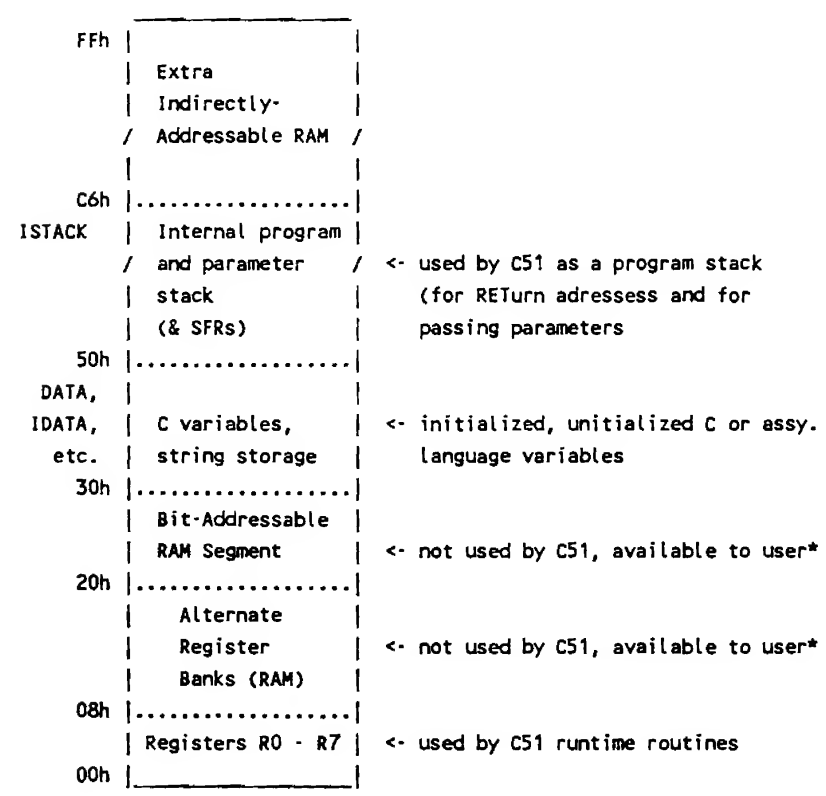


CODE Space Memory Map (Typically PROM)

DATA Space Memory Map (Typically RAM)

8052 INTERNAL MEMORY ORGANIZATION WITH C-51. For the Small (Single-Chip) Model.

This memory map assumes a small model program with a stack segment size of 150 ISTACK). The internal code space memory map is similar to that for the large and medium models (up to 8k bytes of internal code space of the 8052).



* These RAM locations or registers can be accessed directly from a user C program using the input, output, and bit manipulation functions (set_bit, clear_bit, etc.; bit-addressable locations only), or via assembly-language routines called from C.

NOTE: All load addresses are typical, for illustration only. Module size and location is dependedent on program content and linker options.

Index

Error and warning messages are not listed in the Index - see the appropriate Appendix. The TUTORIAL, which covers lots of material found in more detail in other sections of the manual, is not indexed.

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V,W

X

C-51 Release V2.20 Information

The Archimedes C-51 compiler kit will be continually improved.

Compatibility with Digital Research Concurrent DOS, 68K (version 5, in DOS-compatible mode) has been tested (although not thoroughly). Please note that you need to license additional copies of C-51 if you are running the software on more than one machine at a time.

The C-51 Kit has been tested (by Archimedes software developers or emulator vendors) for compatibility with several emulators. Please review next page for details.

"malloc" can currently not be called from more than one piece at a time (i.e. it is not interruptable. All other C-library functions are reentrant).

Remember to return your registration card to receive future product release information. Registered customers will receive updates released within 90 days of original purchase free of charge.

Welcome to Archimedes Microcontroller CI

We are pleased you have chosen the Archimedes C-Z80/64180 Kit for your embedded microcontroller development.

Please make sure that you have received the following:

1. Complete Manual (Ch. 1 -4; App. A - J)
2. License Agreement (end of manual).
3. Three program diskettes.
4. Registration card (in plastic diskette holder).
5. Release Information (colored page).
6. How-To-Get Started (colored page).
7. Installation & System Requirements (colored page).
8. Technical Support Agreement (colored page).
9. 30-day Money-Back Guarantee (colored page).

If any of the above is missing, please contact Customer Service immediately at (415) 567-4010.

The C-Z80/64180 Kit is licensed to you under the terms and conditions of the enclosed Archimedes Single-CPU License Agreement.

Please send in the registration card to assure better technical support and prompt future product information. Your support ID is on the diskette label. By filling in the registration card you are also entitled to free upgrades if a new release is available within 90 days of purchase.

Start out by reviewing the attached release information and the Tutorial section of the manual.

Again, welcome to Archimedes Microcontroller CI.

[Z-80PC V2.22]

TECHNICAL PRODUCT SUPPORT

Archimedes Software provides technical product support for technical product questions about Archimedes C, which you can not work out on your own or find answers to in the manual.

Technical Phone-In Support. Please have the following pertinent information available:

- Version of compiler, linker (displayed at listing).
- Support ID # (see diskette label) available.
- Command lines used by C-compiler and linker.
- Listing of any batch files used to compile,link
- Listing of program modules, library files, linker cross-reference/map list file.
- Listing of CSTARTUP file, XLINK command file.
- If your PC-host runs out of memory, please use "chkdsk" to determine how much memory you have available.

To reach a Technical Support Representative call Archimedes Software at (415) 567-4010 and ask for Technical support. Hours are Monday thru Friday 9 a.m. to 4 p.m. PST. We provide 1 hour of free technical phone support during the first 12 months after you bought the product. (We would obviously not charge you time if you called in a "bug").

The "*Extended Technical Support Agreement*" provides ongoing technical phone support beyond the initial amount provided with original purchase. The cost of this plan is $250 per year and product and provides for up to five hours of technical phone support. Contact us for a copy of this agreement.

Software Issue Reports. This should be used for more complicated technical problems where the software is not performing to its specifications. Please include all pertinent information and clearly identify the specific problem. (Remember to include your support ID #) Send the Report to:

Technical Support, Archimedes Software, Inc, 2159 Union Street, San Francisco, CA 94123.

DOS [illegible]

May be used in .BAT files (IF ERRORLEVEL n) to [illegible] a status value to DOS according to the following:

0	No warnings detected
1	There were warnings detected
2	There were errors detected
3	Fatal error occurred

Diskette #1:

C-51.EXE	C-51 program.
*.H	Header files for use with C-library.

Diskette #2:

CL8051.R03	C-libraries (medium and large model)
CL8051S.R03	C-libraries (small model)
CL8051B.R03	C-libraries (banked model)
FRMWRI.R03	Routine for floating point in (f)printf
FRMWRIS.R03	Same for small memory model.
FRMWRIB.R03	Same for banked memory model.
PUTCHAR.S03	Source of putchar library routine.
GETCHAR.C	Source of getchar library routine.
CSTARTUP.S03	Source of C start-up routine.
CINTERB.R03	Special interrupt routine (banked)
L18.S03	Special file for banking
INTWRI.C	Source for "small printf formatter"
PRINTF.C	Source of library routine.
SPRINTF.C	Source of library routine.
FRMWRI.C	Source of library routine.
DEFEM.INC	Set-up file to use medium/large model.
DEFSC.INC	Set-up file to use small model.
LNK8051.XCL	Linker command file (medium/large)
LNK8051S.XCL	Linker command file (small model)
LNK8051B.XCL	Linker command file (banked)
HEAP.C	File to modify heap size.
SIEVE.C	C source file for example program.
EXAMPLE.*	Files for example program.
CSTARTEX.S03	C start-up routine for example.
PROMSE.S03	File for example.
BIOS1.C	Illustrates in-line functions.

Diskette #3:

XLINK.EXE	Linker.
XLIB.EXE	Librarian.
A8051.EXE	Macroassembler.
ASM1.A03	Assembly demo file.

Installation & System Requirements

System requirements

You need MS-DOS or PC-DOS 2.0 - 3.XX and at least [illegible] memory. The PC must support at least 14 open files at the same time. Modify by editing the DOS-file **config.sys** and change set files=14 (or a larger number) in the **config.sys** file. (See DOS manual). **You are recommended to make this modification prior to using any of the C-51 software.**

Installation.

You can install the software on a hard disk by simply performing the DOS copy commands. **Make back-up copies before you start using the software.** For your convenience, the software is not copy-protected. We trust your integrity in not making illegal copies and only using the software on one machine.

Review the tutorial section for information about modifications of the **cstartup** routine. Also, review section 1.14 Configuration issues in the C-chapter.

Contents of diskettes.

The distribution disks have a number of different files, which are listed on the next page. The following types of different files, distinguished by their file extension, are available:

*.exe	Executable files (e.g. C51.EXE, the compiler)
*.obj	Object files for XLINK and XLIB
*.c	C language source files
*.h	C header files, normally included in C sources
*.inc	Assembly language include files
*.lib	Object module libraries, created by XLIB
*.a51	8051 assembly language source files
*.bat	Batch command files for compiler and linker
*.lst	Output files from compiler, assembler and linker
*.dat	Default configuration files for compiler and linker
*.txt	Various text files with additional information
*.bat	MS-DOS batch files for installation and configuration
*.doc	Files documentation files

How To Get Started

1. Read the Preface at the beginning of the manual.

2. Follow the instructions in Installation & System Management (about 2 pages).

3. Review the tutorial. The code EXAMPLE in the Tutorial (and Appendix I).

4. Review the other parts of the manual as required.

YOU ARE READY TO EDIT, COMPILE AND LINK YOUR FIRST C PROGRAM.

We are pleased you have chosen the Archimedes C-Kit for your [illegible] development.

Please make sure that you have received the following:

1. Complete Manual (Ch. 1 - 5, App. A - G)
2. License Agreement (end of manual)
3. Three program diskettes
4. Registration card (in plastic diskette holder)
5. Release Information (colored page).
6. How-To-Get Started (colored page)
7. Installation & System Requirements (colored page)
8. Technical Support Agreement (colored page)
9. 30-day Money-Back Guarantee (colored page).

If any of the above is missing, please contact Customer Service immediately at (415) 567-4010.

The C-Kit is licensed to you under the terms and conditions of the enclosed Archimedes Single-CPU License Agreement.

Please send in the registration card to assure better technical support and prompt future product information. Your support ID is on the diskette label and on the Technical Support card. By filling in the registration card you are also entitled to free upgrades if a new release is available within 90 days of purchase.

Start out by reading "How-To-Get Started" before using the software.

Again, welcome to Archimedes [illegible]

[illegible]

MONEY-BACK GUARANTEE (P

We believe you will find Archimedes Software's [illegible] C Software Development Kit the right tool for your microcontroller development. In the [illegible] with a 30-day money back guarantee. In the event that does not meet your requirements, please call [illegible] to get a Return Authorization number and enclose the following with a return of the product:

1. Copy of invoice.
2. Complete product including binder, manuals, diskettes, etc. All has to be in a resaleable condition to avoid any return charges.
3. This page signed and filled in. [Call to get Return Authorization # Support ID on diskette label]
4. We would also appreciate some brief notes on the reasons why you are returning the product.

Send within 30 days of purchase (shipping date) to

Attn: Returns
Archimedes Software, Inc.
2159 Union Street
San Francisco, CA 94123

I hereby certify that I and others who have been using the Archimedes C Kit have destroyed any back-up or archival copies of the C Program (Software and Documentation). I have and will use my best efforts to assure that nobody shall do any of this 30-day money-back guarantee.

Signature _____ Date _____

Name _____ Title _____

Company _____ Phone _____

Address _____

Product _____ Return Auth # _____ Support ID _____

TECHNICAL PRODUCT SUPPORT

Archimedes Software provides technical product support for technical product questions about Archimedes C, which you can not work out on your own or find answers to in the manual.

Technical Phone-In Support. Please have the following pertinent information available:

- Version of compiler, linker (displayed at listing).
- Support ID # (see diskette label) available.
- Command lines used by C-compiler and linker.
- Listing of any batch files used to compile,link.
- Listing of program modules, library files, linker cross-reference/map list file.
- Listing of CSTARTUP file, XLINK command file.
- If your PC-host runs out of memory, please use "chkdsk" to determine how much memory you have available.

To reach a Technical Support Representative call Archimedes Software at (415) 567-4010 and ask for Technical support. Hours are Monday thru Friday 9 a.m. to 4 p.m. PST. We provide 1 hour of free technical phone support during the first 12 months after you bought the product. (We would obviously not charge you time if you called in a "bug").

The "*Extended Technical Support Agreement*" provides ongoing technical phone support beyond the initial amount provided with original purchase. The cost of this plan is $250 per year and product and provides for up to five hours of technical phone support. Contact us for a copy of this agreement.

Software Issue Reports. This should be used for more complicated technical problems where the software is not performing to its specifications. Please include all pertinent information and clearly identify the specific problem. (Remember to include your support ID #). Send the Report to:

Technical Support, Archimedes Software, Inc, 2159 Union Street, San Francisco, CA 94123

ARCHIMEDES MICRO-CONTROLLER C

C-8051 V2.20

80202082-38051PC

DOS error-return value

May be used in .BAT files (IF ERRORLEVEL ...) to test return values to DOS according to the following:

0	No warnings detected
1	There were warnings detected
2	There were errors detected
3	Fatal error occurred

[illegible]	
[illegible].EXE	C-[illegible] program
*.H	Header files for use with C-[illegible]
[illegible]	
CL8051.R03	C-libraries (medium and large models).
CL8051S.R03	C-libraries (small model)
CL8051B.R03	C-libraries (banked model).
FRMWRIL.R03	Routine for floating point in (s)printf
FRMWRIS.R03	Same for small memory model
FRMWRIB.R03	Same for banked memory model
PUTCHAR.S03	Source of putchar library routine
GETCHAR.C	Source of getchar library routine
CSTARTUP.S03	Source of C start-up routine
CINTERB.R03	Special interrupt routine (banked)
L1.S03	Special file for banking
PRINTF.C	Source of library routine.
SPRINTF.C	Source of library routine.
FRMWRI.C	Source of library routine.
DEFEM.INC	Set-up file to use medium/large model.
DEFSC.INC	Set-up file to use small model.
LNK8051.XCL	Linker command file (medium/large).
LNK8051S.XCL	Linker command file (small model)
LNK8051B.XCL	Linker command file (banked)
HEAP.C	File to modify heap size.
SIEVE.C	C source file for example program
EXAMPLE.*	Files for example program
CSTARTEX.S03	C start-up routine for example
PULSE.S03	File for example.
8105[illegible].C	Illustrates [illegible]-line functions
[illegible]	
XLINK.EXE	Linker
XLIB.EXE	Librarian.
A8051.EXE	Macro assembler.
A8051*.S03	Assembly demo files

System requirements

You need DOS or PC-DOS 2.0 - 3.X and at least 256K of memory. Floppy disk drive is required for installation purposes. Medium to large programs require 512K memory and a "files=" (or a larger number) in the config.sys file. (See DOS manual). You are recommended to make this modification prior to using any of the C-51 software.

Installation.

You can install the software on a hard disk by simply performing the DOS copy commands. Make back-up copies before you start using the software. For your convenience, the software is not copy protected. We trust your integrity in not making more than 2 copies and only using the software on one machine.

Review the tutorial section for information about modifications of the cstartup routine. Also, review section 1.14 Configuration Issues in the C-chapter.

Contents of diskettes.

The distribution disks have a number of different files, which are listed on the next page. The following type of different files, distinguished by their file extension, are available:

*.exe	executable files (e.g., C51.EXE, the compiler)
*.ovl	overlay files (for XL160 and XL18)
*.C	C language source files
*.h	C header files, normally #included in C sources
*.lib	compiler language include files (source)
*.obj	relocatable and other relocatable object files
*.SD	SIMD assembly language source files
*.A51	assembler source files for compiler and linker
*.lin	default linker file, used by compiler or linker
*.bat	batch command files for compilation or execution
*.doc	ASCII documentation text files

How To Get Started

1. Read this Preface in the beginning of the manual.

2. Follow the guidelines in Appendix A "Getting Started" for your computer.

3. Review the tutorial. The code EXAMPLES in the Tutorial (and Appendix I).

4. Review the other parts of the manual as required.

YOU ARE READY TO EDIT, COMPILE AND LINK YOUR FIRST C PROGRAM.

HP 64000 Support

Please follow these instructions if you are [illegible] emulator with your Archimedes C Compiler [illegible].

1. Link your program using the Archimedes [illegible] generates two different formats. See the Archimedes manual page 3-22. (Review HP-documentation: Hosted Development Systems chapter File Formats and Symbol File Formats for details about the HP-format).

2. If you will use a (HP-supplied) monitor, make sure there are enough of space for the monitor program when you do the linking.

3. If the emulator needs a monitor program, link with the HP linker (on the HP64000 host), placing it at the correct address.

4. Make sure that the HP 64000 system is correctly configured.

5. Download the 'emulation files' created by the Archimedes linker (XLINK) into the HP64000 system.

6. Load the program and the monitor into the Measurement System.

Note: All symbols beginning with "?" are modified so they will begin with "_".

Emulator Support

The C-51 Kit has been tested by Archimedes Software, Inc. and emulator vendors for compatibility with several emulators. (If your emulator supports local symbolic information and line numbers, you need to use the -y option when compiling with C-compiler to generate local symbolic information and line numbers).

Ashling. Use the linker's special Ashling format.

Hewlett-Packard. See attached information.

Huntsville. Use the linker's special Ashling format.

Intel. Use AOMF8051 format.

Metalink. Use SYMBOLIC format.

Microtek (MICE). Use extended tek hex format. Contact Mikrotek for special converter utility.

Nohau. Reads Archimedes symbolic format or AOMF8051 or Ashling format.

Orion. Use Archimedes symbolic format. Choose [F7] Manx/Aztec in Orion's Unilab software.

Signum. Use Signum-supplied converter utility.

Ziltek. Use Ziltek-supplied converter utility.